VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zobrazování procedurálně generovaných planet v reálném čase

Real-Time Procedural Planet Rendering

2017 Jakub Rak

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Rak**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Zobrazování procedurálně generovaných planet v reálném čase
Real-Time Procedural Planet Rendering

Jazyk vypracování: čeština

Zásady pro vypracování:

Dnešní moderní grafické karty díky svému výkonu nabízejí při vykreslování velké možnosti. Cílem této práce je srovnat moderní algoritmy pro generování a následné zobrazování a procházení planet v reálném čase s přihlédnutím k možnostem dnešních grafických karet. Výsledkem práce bude aplikace, která bude umožňovat prohlížení vygenerovaných planet s možnosti procházení a přiblížení až na úroveň lidského pohledu.

1. Nastudujte algoritmy pro generování a vykreslování virtuálních procedurálních planet.
2. Jednotlivé algoritmy teoreticky popište a charakterizujte jejich použití, výhody a nevýhody.
3. Zaměřte se zejména na moderní algoritmy pro procedurální generování.
4. Vytvořte demonstrační aplikaci pro vybrané algoritmy, která bude umožňovat jejich srovnávání (výpočetní náročnost, paměťová náročnost, časová náročnost apod.).
5. Pro implementaci demonstrační aplikaci využijte jazyk C++ s knihovnou OpenGL 4.x.

Seznam doporučené odborné literatury:

[1] D. Shreiner, G. Sellers, J.M. Kessenich and B. M. Licea-Kane, OpenGL Programming Guide, 8th Edition. 2013. 984 p. ISBN 978-0-321-77303-6

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA elektrotechniky a informatiky 1D16 OSTRAVA

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017 .....

Tímto bych rád poděkoval svému vedoucímu Ing. Martinu Němcovi, Ph.D., za poskytnuté odborné rady a za všechen čas, jenž mi takto věnoval.

Abstrakt

Používání procedurálních technik se v dnešní době stále více rozšiřuje zejména díky rychlejšímu hardwaru a potřebě generovat velké množství rozsáhlých světů. V této práci se seznámíme s některými algoritmy pro vykreslení procedurální planety s možností přiblížení až na vzdálenost odpovídající výšce člověka na povrchu planety. Následuje popis tvorby a optimalizace demonstrační aplikace, jež využívá některé popsané algoritmy.

Klíčová slova: procedurální, planeta, generování, vizualizace, OpenGL, GLSL, Perlinův šum, Worleyho šum, optimalizace, compute shader, nevykreslování skrytých ploch.

Abstract

Usage of procedural generation techniques is steadily increasing mainly thanks to the increasingly powerful hardware and desire to generate large quantity of detailed worlds. This thesis describes some of the algorithms used to generate procedural planets with the ability to zoom towards it to the level of human walking on it's surface. Follows a description of the realization and optimization of the demonstration application which uses some of the algorithms described.

Key Words: procedural, planet, generation, visualization, OpenGL, GLSL, Perlin noise, Worley noise, optimization, compute shader, occlusion culling.

Obsah

Seznam použitých zkratek a symbolů

double	–	64 bitová reprezentace desetinného čísla s plovoucí desetinnou čárkou podle standartu IEEE 754
float	–	32 bitová reprezentace desetinného čísla s plovoucí desetinnou čárkou podle standartu IEEE 754
generování segmentu	–	proces, při kterém se na grafické kartě nebo na hlavním procesoru spočítá vzdálenost od středu planety všech vrcholů trojúhelníkové sítě segmentu
mapa	–	pojem mapa je využíván jako synonymum pro texturu
obalové těleso	–	anglicky bouding box nebo bounds, jedná se o primitivní těleso, obvykle kvádr, jenž garantuje, že uvnitř něho jsou všechny vrcholy trojúhelníkové sítě
objekt	–	obecný pojem pro vše, co se dá nějak vykreslit
rozsah segmentu	–	řada bodů definující okraje neboli obrys jednoho segmentu
splat map	–	textura definiující váhu dílčích algoritmů (například výběr biomu) vzhledem k texturovacím souřadnicím
view frustum	–	někdy nazývané jako pohledová pyramida, je 6 matematicky popsaných rovin definujících oblast, jež je viditelná kamerou
vrcholy segmentu	–	vrcholy trojúhelníkové sítě, která je vlastněna segmentem, obrys této trojúhelníkové sítě je stejný jako rozsah segmentu

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Současný svět si již nedokážeme představit bez kvalitně zpracovaných filmů a her. Dnešní divák je náročný a tvůrci filmů a her jsou nuceni používat nejmodernější techniky zpracování, aby udrželi krok v konkurenčním prostředí. Důležitým prvkem při vývoji her či filmů je tvorba rozsáhlého a zdánlivě realistického terénu. K tomu lze využít řadu metod a technik, mezi nimiž má své místo i procedurální generování [1].

Procedurální generování se často používá pro tvorbu obsahu, hovoříme tedy o procedurálním generování obsahu (anglicky Procedural Content Generation, zkráceně PCG). Procedurální generování obsahu je definováno jako algoritmické tvoření obsahu pomocí omezeného nebo nepřímého zásahu návrháře [2]. Jinak řečeno, procedurální generování obsahu využívají programy, které dokáží obsah generovat samy nebo za účasti návrhářů.

Pod pojmem obsah jsou myšleny objekty a tělesa, které má smysl takto procedurálně generovat, v závislosti na kontextu to mohou být například hrací plochy, terén, povrch planet, herní pravidla, textury, příběhy, předměty či hudba. Slova procedurální generování implikují, že se využívají procedury (neboli funkce, metody či algoritmy), které tvoří obsah a mohou být spuštěny na osobním počítači.

Asi nejpoužívanějším příkladem procedurálního generování je pseudonáhodná funkce random, která se inicializuje pomocí programátorem definovaného čísla (anglicky zvané seed). Pomocí stejného seed lze dosáhnout opakovatelných deterministických výsledků. Pseudonáhodnost znamená, že vygenerovaná data vypadají zdánlivě náhodně, ovšem ve skutečnosti jsou algoritmem vygenerována deterministicky.

V dnešní době existuje již řada vysoce specializovaných programů (Terragen [3], E-On Vue [4], World Machine [5], atd.), které se v profesionálních sférách běžně využívají. Koupě a použití zmíněných programů snižuje dlouhodobé náklady a čas na produkci, neboť návrháři nemusí terén (a jiné objekty) vytvářet ručně, ale mohou je v nástrojích rychleji vygenerovat.

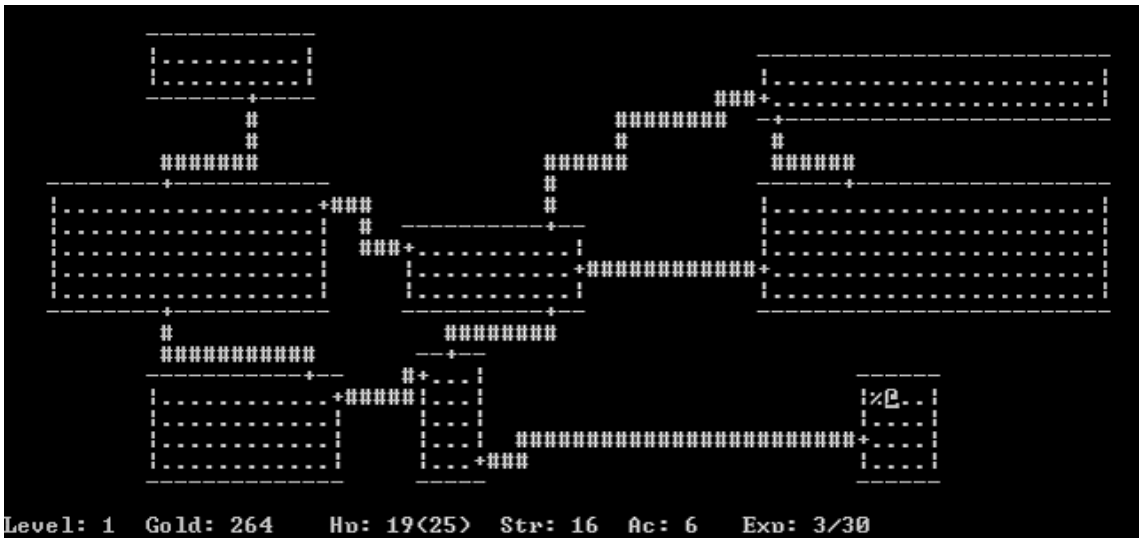
Další výhodou procedurálního generování je možnost variability vygenerovaných dat pomocí relativně malé změny parametrů (například seed a jiné). Procedurální techniky jsou na vzestupu i díky neustálému růstu výkonu a paralelizace hardwaru.

V této práci se seznámíme především se základy procedurálního generování zaměřeného hlavně na povrch planet. Popíšeme několik algoritmů a realizaci demonstrační aplikace. Dále, neboť chceme planetu vykreslovat v reálném čase, popíšeme a použijeme řadu optimalizačních technik. Tato práce je založena na předcházející bakalářské práci [6] s názvem Procedurální generování terénu.

1.1 Historie

Historicky se procedurální generování obsahu začalo využívat v ASCI hrách hlavně kvůli velice limitované paměti tehdejších počítačů. Například ASCI hra Rogue [7], vytvořena v roce 1980, využívala procedurální generování ke konstrukci dungeonu. V dnešní době jsou hry typu Rogue

mezi vývojáři velmi populární, neboť jim dovolují zaměřit se hlavně na herní mechaniky místo toho, aby trávili čas tvořením herních světů [8].



Obrázek 1: Hra Rogue [7].

Podobné využití našly techniky procedurálního generování obsahu i v prvních 3D hrách. Programátoři se stále potýkali s problémem nedostatku paměti. A proto byli nuceni nechat části herních ploch vygenerovat procedurálně až v momentě, kdy je hráč může spatřit.

S nástupem lepšího hardwaru a kompaktních disků [9] (zkráceně CD, první CD byly vydány v roce 1982), již nebylo využití procedurálních technik nezbytné. Možnost ručně vytvořit všechny modely a mapy dopomohlo k lepší unikátnosti obsahu her. Ovšem i zde je problém, návrhář může vytvořit pouze limitované množství modelů. Jedno z řešení je kombinovat procedurálně generované modely a návrhářem vytvořené textury, jako je tomu například v SpeedTree [10], kde jsou textury listů načítány ze souboru, zatímco základní tvar stromu je generován procedurálně. Další možností je návrhářem vytvořené objekty částečně procedurálně poupravit, například deformovat, čímž se dosáhne zdánlivé variace.

Extrémním příkladem je hra Elite [11] (vydaná v roce 1984), kde hráč originálně mohl navštívit až 2^{48} (přibližně 282 biliónů) galaxií, z nichž každá má 256 solárních systémů. Vydavatel se ovšem bál, že by tomu potenciální zákazníci nevěřili a tak byl nakonec počet galaxií zredukován na pouhých 8.

Další ukázkou procedurálního generování obsahu je hra The Elder Scrolls: Arena [12] vydaná v roce 1994, kde mezi městy byla několik stovek kilometrů rozsáhlá procedurálně vygenerovaná hrací plocha (především lesy).

Obrázek 2: The Elder Scrolls: Arena [12].

1.2 Aktuální stav

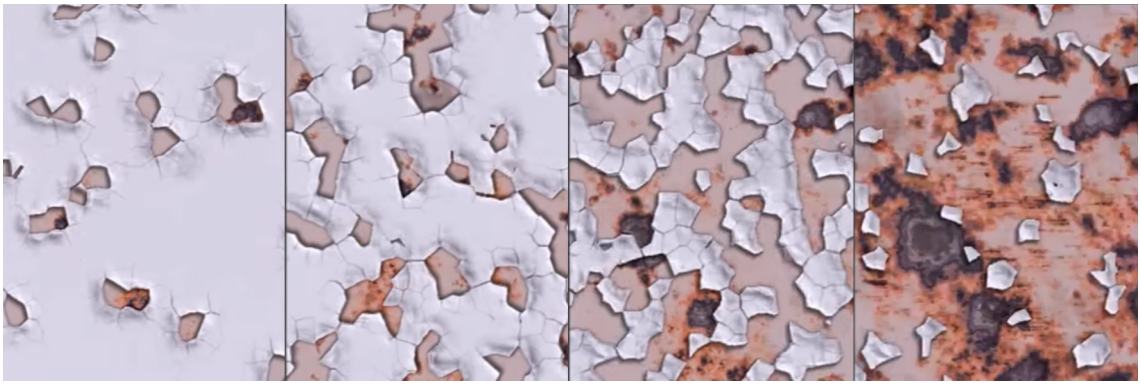
Často lze procedurální generování, i syntézu zvuků a textur, najít v demoscéně. Demoscéna (anglicky demoscene) [13] je fenomén na pomezí digitálního umění, jež je prezentováno na počítačích. Příkladem z demoscény je hra jménem .kkrieger [14], která zabírá pouze 96 kb, všechny modely, textury a zvuky jsou v této hře vygenerovány a syntetizovány procedurálně.

V úvodu zmíněné nástroje (Terragen [3], E-On Vue [4], World Machine [5], atd.) se již využívají řadu let, využívají se ovšem pouze jako nástroj, který vygeneruje terén jednou a koncová aplikace jej poté pouze načítá. Tento přístup je dobrý například pro filmový průmysl, kde návrháři můžou vygenerovanou scénu libovolně upravovat. Ve filmovém průmyslu je také jednodušší zajistit dostatečně výkonné a paralelizované výpočetní zdroje.

Pokud ovšem chceme totéž zobrazit na běžném osobním počítači, je lepší z důvodu menšího úložiště a výpočetních zdrojů, zvolit odlišný přístup, a sice vygenerovat pouze to, co je potřeba za běhu aplikace. Totéž platí, pokud chceme dosáhnout velkého počtu (několik biliónů) velice detailních planet, vygenerovat a zobrazit je všechny najednou v jeden okamžik. Toto není z paměťového hlediska možné.

Tento problém se řeší tak, že aplikace si planety podle potřeby generuje sama. Algoritmy generující planety tedy přesuneme z externích zdrojů do aplikace samotné.

Existují nástroje (například: Quixel Suite [15], Speed Tree [10], Substance Designer [16], atd.), jejichž výstup částečně popisuje procedurální složku objektů a tak lze v aplikaci využít jejich procedurální variace.



Obrázek 3: Procedurální textura zdi v Substance Designer s různými hodnotami stáří [16].

Běžný člověk se v dnešní době s tímto způsobem generování objektů může setkat hlavně ve hrách (například hry Kerbal Space Program, No Mans Sky, Elite Eangerous, nebo Star Citizen).

2 Teorie pro procedurální generování planety

Procedurální generování planet v malém detailu je relativně jednoduché, planetu stačí pouze jednou vygenerovat a zobrazit. Tento přístup je dostatečný pouze v případě, že se na planetu budeme dívat z vesmíru a z velké vzdálenosti. Pokud se k planetě přiblížíme, uvidíme, že není dostatečně detailní. Naším požadavkem je, aby planeta byla dostatečně detailní jak při pohledu z vesmíru, tak i z pohledu člověka stojícího na povrchu. Musíme tedy využít řadu algoritmů a optimalizačních technik, které nám dovolí detail planety měnit dynamicky podle potřeb na vybraném místě.

2.1 Náhodnost

Často zmiňovaným pojmem v této práci je náhodnost, či pseudonáhodnost. Teoreticky vše podléhá pevně daným fyzikálním zákonům, můžeme tedy říct, že opravdová náhodnost v pravém smyslu slova neexistuje. Náhodu lze charakterizovat jako souhrn drobných, ne zcela zjistitelných, nebo vůbec nezjistitelných vlivů [17]. Variací náhodnosti je pseudonáhodnost, pseudonáhodnost lze prakticky a deterministicky předvídat. Deterministická pseudonáhodná sekvence dat je obvykle ze vstupních dat vypočítána nějakým algoritmem, použitá vstupní data se nazývají seed (z anglického slova semínko).

2.2 Generátor pseudonáhodného čísla

Asi nejznámějším příkladem procedurálního generování je výpočet pseudonáhodného čísla random. Ostatní algoritmy využívané v procedurálním generováním fungují na podobném principu. Je zde základní inicializační hodnota (seed), od které se potom počítají hodnoty ostatní. Perlinův a Worleyho šum je v podstatě pouze interpolací mezi hodnotami, které jsme vygenerovali pomocí generátoru pseudonáhodných čísel random.

```
class Random {
  uint x;
  void SetSeed(uint seed) {
    x = seed;
  }
  uint GetNext() {
    x = x * 61061 + 1;
    return x;
  }
}
```

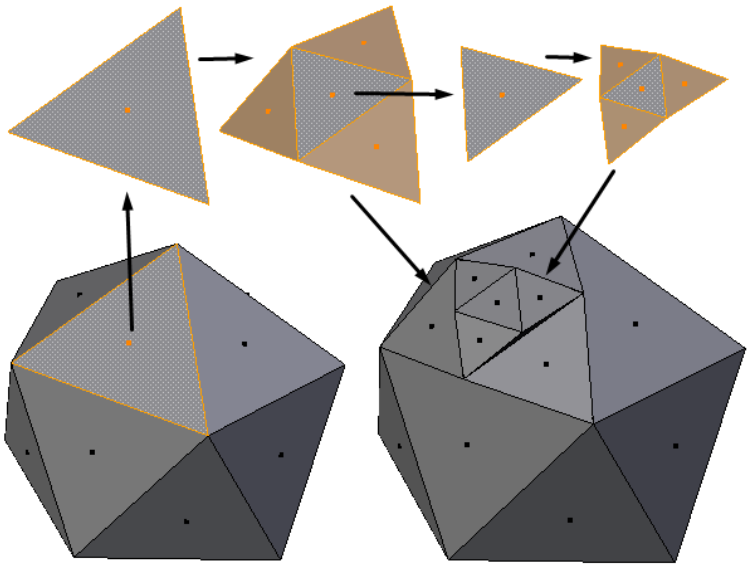
Výpis 1: Příklad jednoduché třídy Random.

2.3 Rozdělení planety na stromovou strukturu segmentů

Pokud máme planetu o velikosti Země (kde jednotka v našem souřadném systému reprezentuje jeden reálný metr), je na běžném hardwaru nemožné v reálném čase načíst a vykreslit najednou celý její povrch v detailu přijatelném i pro člověka na povrchu. Rovněž je zbytečné vykreslovat celou planetu za pomocí milionů trojúhelníků, když je z nich ve výsledku na obrazovce viditelných pouze několik jednotek pixelů. Proto je potřeba mít schopnost detail planety na libovolném místě planety zvětšit. Toto zvětšování detailu se obvykle anglicky nazývá Chunked LOD (kde LOD znamená Level Of Detail) [18]. Planetu složíme ze segmentů, kde proces zvětšování detailů bude probíhat tak, že vybraný segment rozdělíme na potomky, a tento proces rekurzivně opakujeme dokud nedosáhneme chtěného detailu na vybraném místě planety. V podstatě generujeme, načítáme a zobrazujeme pouze ty segmenty, které jsou nezbytně nutné.

Povrch planety je rozdělen na segmenty. Tyto segmenty jsou ve stromové struktuře, kde na začátku této struktury jsou kořenové segmenty. Každý segment má svůj rozsah (neboli obrys či tvar, anglicky range), jenž definuje, kde na planetě se nachází. Každý segment se poté může rozdělit na další dceřiné segmenty (children), proces rozdělení segmentu vytvoří další segmenty, jejichž společný rozsah je stejný, jako rozsah rodičovského segmentu. Toto rekurzivní rozdělování segmentů znamená, že každý segment může mít své rodiče a svoje potomky. Každý segment má také svojí trojúhelníkovou síť (mesh), jejíž obrys kopíruje rozsah segmentu. Tuto trojúhelníkovou síť je potřeba vygenerovat.

Následující obrázek a ukázka třídy pro segment by mohla tento koncept segmentů lépe objasnit.



Obrázek 4: Ukázka rekurzivního rozdělení segmentů, kde kořenové segmenty jsou vytvořeny z dvacetistěnu.

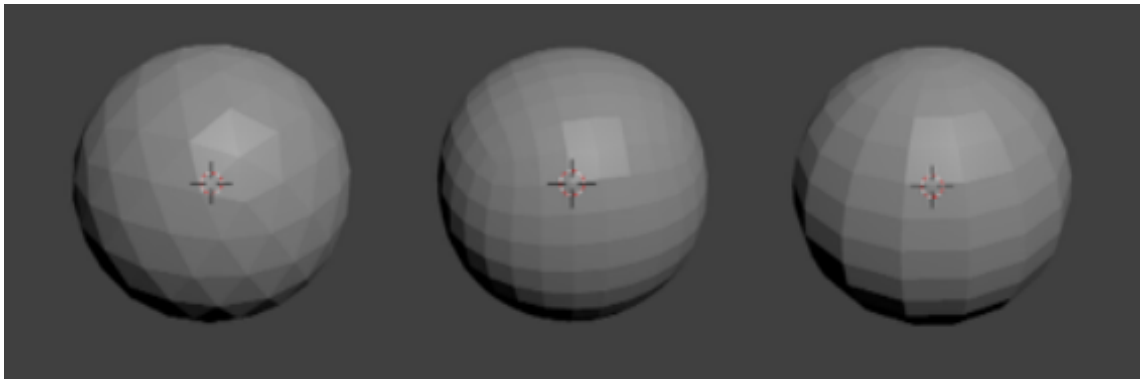
```
class Segment {
  vec3[] range;
  Mesh* mesh;
  Segment* parent;
  Segment*[] children;
}
```

Výpis 2: Ukázka vlastností třídy pro segmenty.

2.4 Tvar kořenových segmentů

Jedno z klíčových rozhodnutí, od něhož se odvíjí ostatní algoritmy, je tvar kořenových segmentů. Tvar dceřiných segmentů je odvozen z kořenových segmentů, neboť dceřiné segmenty vzniknou jeho rozdělením. Je žádoucí, aby všechny segmenty ve stejné generaci měly stejně velký rozsah a aby vzdálenosti mezi vrcholy jejich trojúhelníkových sítí byly co nejvíce shodné. Tím se nám zjednoduší následné algoritmy, protože nemusíme počítat s odchylkami plynoucími z vybraného tvaru a povrch planet bude na všech místech stejně detailní. Rozsahy kořenových segmentů můžeme získat z topologie aproximace kulové plochy. Máme na výběr z několika druhů aproximace kulové plochy, můžeme si vybrat například mezi těmito třemi [19]:

- UV kulová plocha je kulová plocha, jejíž vrcholy jsou rozprostřeny dle zeměpisné šířky a výšky. Nevýhodu je, že na pólech je rozlišení největší, zatímco na poledníku je rozlišení nejmenší. Velikost kořenových segmentů se mění v závislosti na zeměpisné šířce.
- Kulová plocha z krychle je vytvořena z detailní trojúhelníkové sítě ve tvaru krychle, jejíž vrcholy mají normalizovanou vzdálenost od středu. Tato topologie je lepší než UV kulová plocha. Avšak zejména na hranách krychle lze vidět odchylky v rozprostření vrcholů. Pokud je velikost planety dostatečně velká, pak z pohledu člověka na planetě již nejsou odchylky ve vzdálenosti vrcholů tak viditelné.
- Kulová plocha z dvacetistěnu je koule, jež je tvořena dvaceti stejnými trojúhelníky. Rozsahy kořenových segmentů jsou tyto trojúhelníky. Jelikož jsou tyto trojúhelníky naprosto stejné, všechny segmenty ve stejné generaci mají taktéž stejný tvar.



Obrázek 5: Kulová plocha z dvacetistěnu (anglicky icosahedron), Kulová plocha z krychle, UV kulová plocha [19].

Je zřejmé, že pro naše účely mají kořenové segmenty vytvořené z dvacetistěnu nejlepší vlastnosti. Zajímavostí je, že některé komerční hry využívají tvar kořenových segmentů odvozený od kulové plochy z krychle, nebo od UV kulové plochy. Pravděpodobně je to z toho důvodu, že práce se čtverci je jednodušší, více intuitivní a také umožňuje lepší a jednodušší využití textur (například pro výškové či normálové mapy).

2.5 Selektivní zvětšování detailu planety

Jelikož máme možnost rozdělit libovolný segment, je potřeba algoritmu, jenž vybere ideální segmenty pro rozdělení. Naším cílem je, aby segmenty nejblíž ke kameře byly nejvíce detailní, tento požadavek lze formulovat jako: chceme, aby velikost každého segmentu na obrazovce nebyla větší, než definovaná hodnota. V práci "Rendering Very Large Very Detailed Terrains" [18] byl pro tento účel využit následující algoritmus:

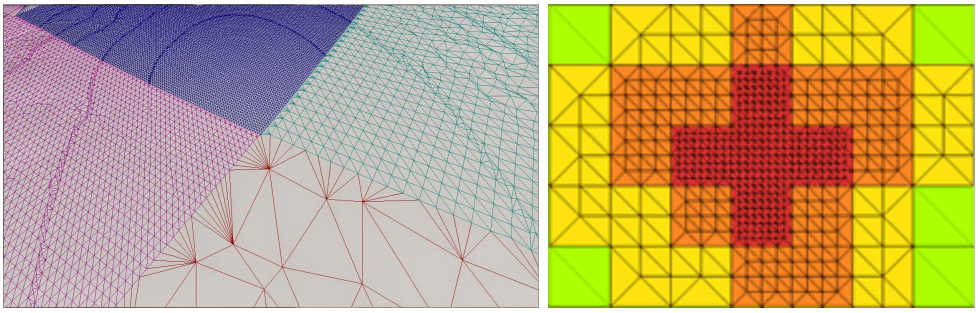
```
detail_level_select (node) {
    d = distance from viewpoint to node
    if d < delta_c (node) * C then
        if children not in memory then
            request loading of children into memory
            select node for rendering
        else
            sort children according to distance
            for each child of node
                detail_level_select (child)
            else
                select node to be rendering
```

Výpis 3: Pseudokód pro výběr detailu segmentů.

V tomto algoritmu se pro každý segment spočítá priorita, pokud tato priorita převyšuje určenou hodnotu, segment se rozdělí na potomky (zvýší se detail segmentu). Výše zmíněný algoritmus ovšem předpokládá, že trojúhelníková síť segmentů je uložena předem a my pouze vybíráme její dostačující aproximace pomocí odchylky této aproximace od nejdetailnější trojúhelníkové sítě. Tento algoritmus je tedy potřeba pro naše účely upravit.

Při selektivním zjemňování detailu může nastat situace, kde sousedící viditelné segmenty jsou z rozdílných generací. Máme tedy dva sousedící segmenty s rozdílným detailem a velikostí, na jejichž hranici se poté mohou vyskytnout artefakty ve formě prasklin (anglicky cracks). Tyto praskliny lze skrýt či eliminovat například těmito způsoby [18]:

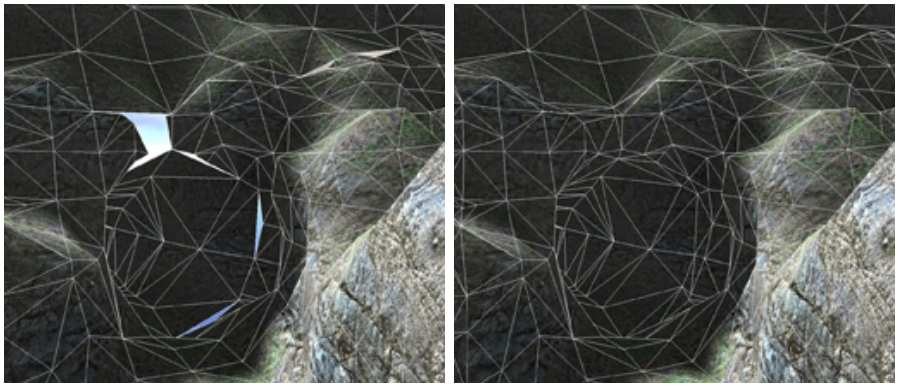
1. Sešití (anglicky sewing) - tato metoda projde vrcholy na hranách dvou segmentů a podle potřeby je přesune či odstraní. Nevýhodou je, že tuto metodu je potřeba použít pokaždé, když se změní sousedící segmenty.



Obrázek 6: Ukázka sešití sousedících segmentů [1].

2. Ucpání (anglicky seam) - metoda vytvoří další trojúhelníkovou síť, která přesně kopíruje tvar praskliny a celou ji zakryje. Stejně jako u předchozí metody, trojúhelníková síť pro zakrytí prasklin je závislá na obou sousedících segmentech, trojúhelníkovou síť je tedy potřeba znovu vypočítat pokaždé, když se sousedící segmenty změní. U voxelových dat existuje pro ucpání voxelových segmentů Transvoxel^{TM} algoritmus.

[1] Převzato z: `http://vterrain.org/LOD/Papers/`



Obrázek 7: Transvoxel^{TM} algoritmus [20].

3. Sukně (anglicky skirts) - je nejjednodušší metoda a také jediná metoda která se nemusí znova vypočítat, pokud se sousedící segmenty změní. Tato metoda se také běžně používá ve hrách (například: Fallout 4). Metoda vytvoří na hranicích segmentů převisy (neboli sukně), jenž vizuálně zakryjí většinu prasklin.

2.6 Formát dat popisující planetu

Většina algoritmů bude pracovat s popisem planety v pevně daném formátu, proto je nutno vybrat formát, jenž je pro naše účely nejpřijatelnější. Povrch planety lze zapsat několika formáty, každý ze zmíněných formátů má své výhody i nevýhody. Například si můžeme vybrat mezi těmito formáty:

- Trojúhelníková síť (anglicky triangle mesh, nebo jen mesh) je formát, se kterým pracuje grafická karta, všechny ostatní možnosti se v podstatě na konci zpracování musí převést do trojúhelníkové sítě. Pokud pracujeme s trojúhelníkovou sítí od začátku, naše algoritmy jsou jednodušší, neboť již nemusí ostatní formáty převádět do trojúhelníkové sítě. Nevýhodou je o něco složitější manipulace, pokud chceme například změnit výšku planety v určeném místě, musíme nejdřív najít správné vrcholy.

- Výšková mapa je dvojrozměrná textura, která reprezentuje nadmořskou výšku nebo vzdálenost od středu planety na základě zeměpisné šířky a výšky. Zřejmou nevýhodou tohoto formátu je sférická distorze textury způsobená mapováním textury na kulovou plochu. Je zde viditelná rozdílná hustota dat na základě zeměpisné výšky. Výhodou je jednoduchý převod do trojúhelníkové sítě, jednoduchá modifikace a malé paměťové nároky.

- Voxelová reprezentace definuje hustotu planety v trojrozměrném prostoru, z paměťového hlediska lze voxelová data uložit například do trojrozměrné textury. Nevýhodou je složitější převod do trojúhelníkové sítě a výhodou je velmi jednoduchá manipulace s daty. Tento formát se nejčastěji využívá ve hrách, kde hráč může planetu libovolně měnit (například hry: Space Engineers, Planet Nomads, No Man's Sky a jiné).

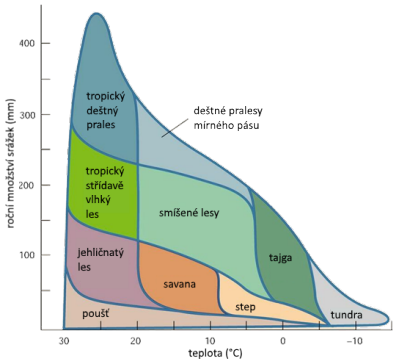
V některých případech není vhodné použit pouze jeden druh reprezentace povrchu planety. Abychom dosáhli jednoduší manipulace a rychlejší generování, lze jednotlivé formáty dle potřeb kombinovat. Někdy je také potřeba mezi formáty dat převádět, používají se k tomu různorodé algoritmy, z nichž jeden, pro naivní převod výškové mapy do voxelové reprezentace může být následující algoritmus.

```
for(int x = 0; x < voxel_data.size.x; x++)
    for(int y = 0; y < voxel_data.size.y; y++)
        for(int z = 0; z < voxel_data.size.z; z++)
            if(height_map[x, y] < z * 255)
                voxel_data[x, y, z] = 1;
            else
                voxel_data[x, y, z] = 0;
```

Výpis 4: Kód pro naivní převod výškové mapy do voxelové reprezentace.

2.7 Biomy

Zatím jsme popisovali pouze topologickou strukturu planety, to je samo o sobě důležité. Ovšem to, co uživatel vidí, jsou textury povrchu planety a objekty, jež se na povrchu nacházejí. Pro tento účel se používá koncept biomů (anglicky biomes, nebo climates). Například podle zeměpisné výšky, nadmořské výšky a vzdálenosti od oceánů se vypočítá teplota a vlhkost. A na základě teploty a vlhkosti se vybere z několika definovaných biomů. Každý biom má definované algoritmy, objekty a textury, jež se na povrch planety aplikují. Obvykle se používá kontrolní textura, která na základě některých výše zmíněných hodnot biom vybere, příkladem takovéto kontrolní textury je Whittakerův diagram [21], který na základě teploty a množství srážek popisuje biomy Země. Pod pojmem biom si můžeme představit například tundru, les, oceán či poušť.



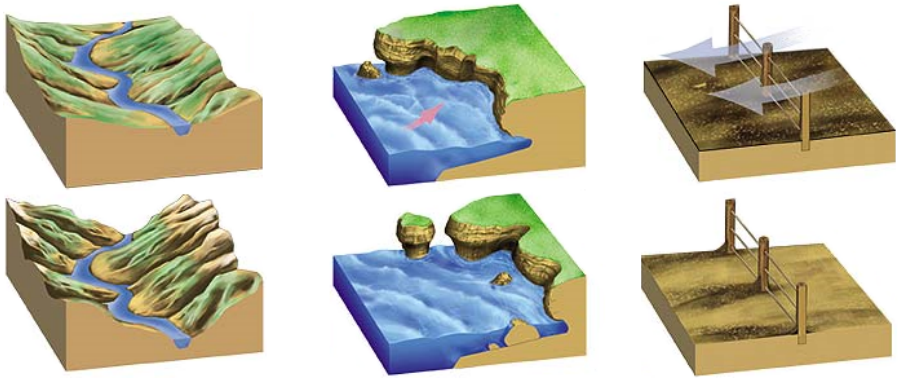
Obrázek 8: Whittakerův diagram [21].

2.8 Eroze

Terén v reálném světě se neustále mění, jedním z procesů změn je například eroze. Eroze je přirozený povrchový proces, jenž za pomocí vnějších sil odstraňuje nebo rozpouští horninu a přesouvá ji na jiné místo.

Existuje několik druhů erozí:

- Gravitační eroze - síla gravitace způsobuje svahové pohyby hornin a sedimentů.
- Větrná eroze - činností větru dochází k přenosu materiálu, síla větrné eroze je závislá na síle a úhlu dopadu větru.
- Vodní eroze
  - Eroze působením ledu
  - Řeky a potoky
  - Pobřežní eroze
  - Povodně



Obrázek 9: Některé druhy eroze, zleva: vodní eroze řek, pobřežní eroze, větrná eroze [2].

Druhy eroze, které jsou viditelné na největší ploše země, jsou vodní a větrná eroze, tyto druhy eroze bychom tedy nejraději simulovali. [22] Aby terén byl co nejrealističtější, lze například na výškové mapě terénu nasimulovat několik cyklů vodní eroze. Naším cílem je, aby terén vypadal pěkně, koncového uživatele pravděpodobně nezajímá, zda eroze je či není dokonalou simulací reality. Více reálná simulace eroze navíc využívá většího množství výpočetních prostředků, které můžeme lépe využít na jiných místech [23]. Pro potřeby eroze lze využít například algoritmy z práce "Interactive Terrain Modeling Using Hydraulic Erosion" [24].

[2] Upraveno a převzato z: `https://www.britannica.com/science/erosion-geology`

2.9 Vícestupňové skládání dat popisující povrch planety

Výška popisující planetu se může složit z několika stupňů detailu, kde každý tento stupeň může mít jiný formát a odlišný algoritmus pro zobrazení. Data popisující povrch planety lze například rozdělit do těchto stupňů:

1. Základní výšková mapa planety, výšková mapa definující základní výšku (tvar) planety. Tuto výškovou mapu lze získat například vygenerováním v naší aplikaci, v externím nástroji jakým je World Machine [5], nebo stáhnout z webových stránek NASA. Výškovou mapu lze uložit a návrhář ji může podle potřeb upravovat.

2. Podle požadavku nasimulovat menší detaily ve formě malých kráterů a kopců pomocí šumových funkcí až ve chvíli, kdy je potřeba.

3. Nakonec při vykreslení každého snímku nasimulovat nejmenší detaily v teselačním shaderu podle výškových map z biomů.

2.10 Posuvný registr s lineární zpětnou vazbou

Jeden z nejpřímočařejších způsobů generování pseudonáhodného čísla nebo bitu je posuvný registr s lineární zpětnou vazbou (anglicky: Linear Shift Feedback Register), dále jen LFSR. Kvůli své nenáročnosti byl s oblibou využíván ve starších 8 bitových hrách [25]. Díky jeho hardwarové jednoduchosti ho lze také jednoduše poskládat z běžně dostupných součástek, jeho hardwarová variace se běžně používá v bezdrátovém přenosu dat. V procedurálním generování se LFSR dá použít například pro zjištění, zda na pozici má či nemá být vygenerován objekt.

```
int lfsr = 1;
for(int y=0; y<200; y++) {
  for(int x=0; x<200; x++) {
    bool bit = ((lfsr >> 0) ^ (lfsr >> 1) ^ (lfsr >> 2) ^ (lfsr >> 3)) & 1;
    lfsr = (lfsr >> 1) | (bit << 15);
    if(bit) set black at x,y
    else set white at x,y
  }
}
```

Výpis 5: Kód pro generování bitového LSFR šumu.

3 Metody urychlení a optimalizace

Jedním z hlavních úkolů při vývoji jakékoliv aplikace vykreslující v reálném čase je zajistit stabilní a dostatečný počet překreslení snímků za sekundu. V dnešní době již existuje řada metod a algoritmů, které se běžně používají a jsou dostupné. Teoreticky by se dalo říct, že čím více jich použijeme tím budou naše výsledky lepší, prakticky ale záleží na typu aplikace a na jakém hardwaru bude aplikace běžet. Jsou rozdíly mezi hardwarovou architekturou osobního počítače a herní konzole, optimalizaci kterou použijeme pro osobní počítač by u herní konzole mohla mít opačný výsledek. [26]

3.1 Nevykreslování skrytých ploch

Někdy také anglicky nazývané jako Hidden Surface Determination, Visible Surface Determination nebo Occlusion Culling. Je to skupina technik, která se zabývá problematikou determinace skrytých ploch. Skrytými plochami jsou zde myšleny objekty nebo polygony, jenž uživatel nevidí. Hlavním účelem je tyto skryté plochy nevykreslovat a tím ušetřit výkon jak na hlavním procesoru, tak na grafické kartě. Je ovšem nutno použité algoritmy dostatečně optimalizovat a využít jen tehdy, pokud nám opravdu výkon ušetří. Máme na výběr z řady možností, které lze podle potřeb také kombinovat: [27]

3.1.1 Hardware Occlusion Queries

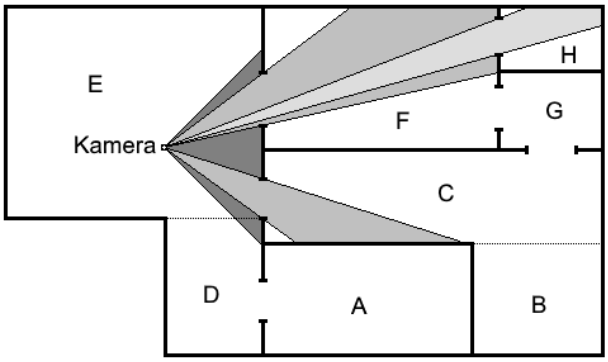
Jednou z nejočekávanějších hardwarových funkcí grafických karet byly Hardware Occlusion Queries [28] [29]. Tato hardwarová funkce nám umožňuje zjistit, zda objekt, který chceme vykreslit, bude ve výsledném snímku viditelný. Pokud se takto dotážeme například na obalové těleso několika objektů, zjistíme, zda má smysl je vykreslit. Výhodou je, že se takto můžeme dotázat na libovolně složitý objekt. Ovšem nevýhodou je cena, kterou za každý takovýto dotaz platíme, naivní využití Hardware Occlusion Queries nám aplikaci naopak zpomalí. Pro efektivní využití Hardware Occlusion Queries je tedy potřeba mít scénu uloženou do hierarchické struktury, která nám umožní chytře využít výsledky z předchozích snímků.

3.1.2 Softwarový z bufer rasterizer

Jedná se o metodu, jež napodobuje hardwarový z bufer. Každé těleso do tohoto softwarového z buferu vykreslí svoji aproximaci, takzvaný occluder [30]. Tento occluder je velice zjednodušená trojúhelníková síť objektu, jež říká, která místa reálného objektu budou vždy zakrývat ostatní objekty. Každé těleso poté zjistí, zdali hloubka jeho obalové krychle je před, či za hodnotami v našem softwarovém z buferu a na základě této informace se rozhodne, zdali se vykreslí. Nevýhodou tohoto přístupu je potřeba tvorby occluderu. I zde lze využít řadu optimalizací. Je možné například místo interpolace hloubky právě rasterizovaného trojúhelníku tuto hloubku vykreslovat konstantní, kde tato konstantní hodnota je definovaná nejvzdálenějším vrcholem.

3.1.3 Portály

Anglicky Portals, Portal Culling, nebo Portals and Areas, je metoda, u které návrhář při tvorbě scény definuje konvexní polygony (takzvané portály), které spojují definované prostory (z anglického areas). Pokud je viditelný portál, pak jsou viditelné i prostory s ním spojené. Tento algoritmus je ve své podstatě vylepšenou specializovanou verzí algoritmu Frustum Culling, neboť pro každý portál se vytváří matematicky popsaný výřez z pohledové pyramidy [31].



Obrázek 10: Ukázka algoritmu Portal Culling: prostory C, D, E, F a H jsou viditelné skrz portály [3].

3.2 Využití více logických jednotek procesoru

Samozřejmostí je snaha využít všechny zdroje počítače. Toho lze dosáhnout například přesunutím náročných výpočtů do samostatných vláken procesoru. Ovšem je potřeba se rozhodnout, co je pro naší aplikaci důležitější, větší počet snímků překreslení za sekundu nebo minimalizace doby, kterou trvá něco spočítat (procedurálně vygenerovat), maximalizovat obojí kvůli pevně danému maximálnímu výkonu počítače nejde. Dává smysl počet vykreslení snímků za sekundu limitovat na 60, neboť většina monitorů nedokáže více využít. Zbylý výkon se poté využije v ostatních částech aplikace.

3.3 Virtuální textury

Pokud chceme aby každý segment vypadal jinak než ostatní, lze toho docílit také složením více textur. Máme tedy kontrolní texturu (takzvanou splat mapu) a dílčí textury, splat mapa poté definuje jakou dílčí texturu použít a na jakém místě (dílčí mapy se obvykle používají s rozdílným měřítkem). Virtuální textury jsou výsledkem procesu, kdy se výsledná textura vykreslí (složí) jednou místo toho, aby se vykreslovala (skládala) při každém snímku na grafické kartě. Virtuální textury nám tedy hlavně šetří výkon a také nám umožňují využít většího množství textur a jiných algoritmů, které bychom jinak z výkonového hlediska použít nemohli.

[3] Upraveno a převzato z: `http://softpixelengine.sourceforge.net/forum/viewtopic.php?f=9&t=91`

4 Popis realizace demonstrační aplikace

Cílem demonstrační aplikace bylo naprogramovat generování planety, na kterou je možno pohlížet jak z vesmíru, tak z pohledu člověka stojícího na jejím povrchu.

Planeta je rozdělená na segmenty. Segmenty se podle potřeb rekurzivně selektivně rozdělují a tím se dynamicky mění detail planety, pro potřeby výběru detailu se využívá rozsah segmentu. Nadmořská výška planety je generována pomocí několika kombinací šumových funkcí. Aplikace byla naprogramována tak, aby se snažila udržet 60 překreslení snímků za sekundu, aplikace tedy podle potřeb snižuje počet segmentů, které za jednu minutu vygeneruje. Protože aplikace byla tvořena spolu s vlastním vykreslovacím enginem, bylo potřeba tento vykrajovací engine také optimalizovat.

4.1 Testování a výsledky

Aplikace byla testována na třech počítačích s konfiguracemi uvedenými v následující tabulce. Výsledky měření se mohou na jiných konfiguracích lišit. Při všech testech byla demonstrační aplikace ve výchozím rozlišení vykreslovacího okna 1400 na 900 pixelů. Segmenty byly nastaveny tak aby generovali celkem 3240 vrcholů (což je 80 vrcholů na hraně).

	hlavní procesor	grafická karta
PC1	Core 2 Quad Q9650	NVIDIA GTX 970
PC2	FX-6300 VISHERA	NVIDIA GTX 760
PC3	Intel(R) Core(TM) i5-3570 CPU @ 3.40GHZ	NVIDIA Quadro 600
PC4	Intel(R) Core(TM) i7-5820 CPU @ 3.30GHZ	NVIDIA GeForce GTX 1070

Tabulka 1: Konfigurace počítačů, na nichž byla demonstrační aplikace testována.

4.2 Použité technologie

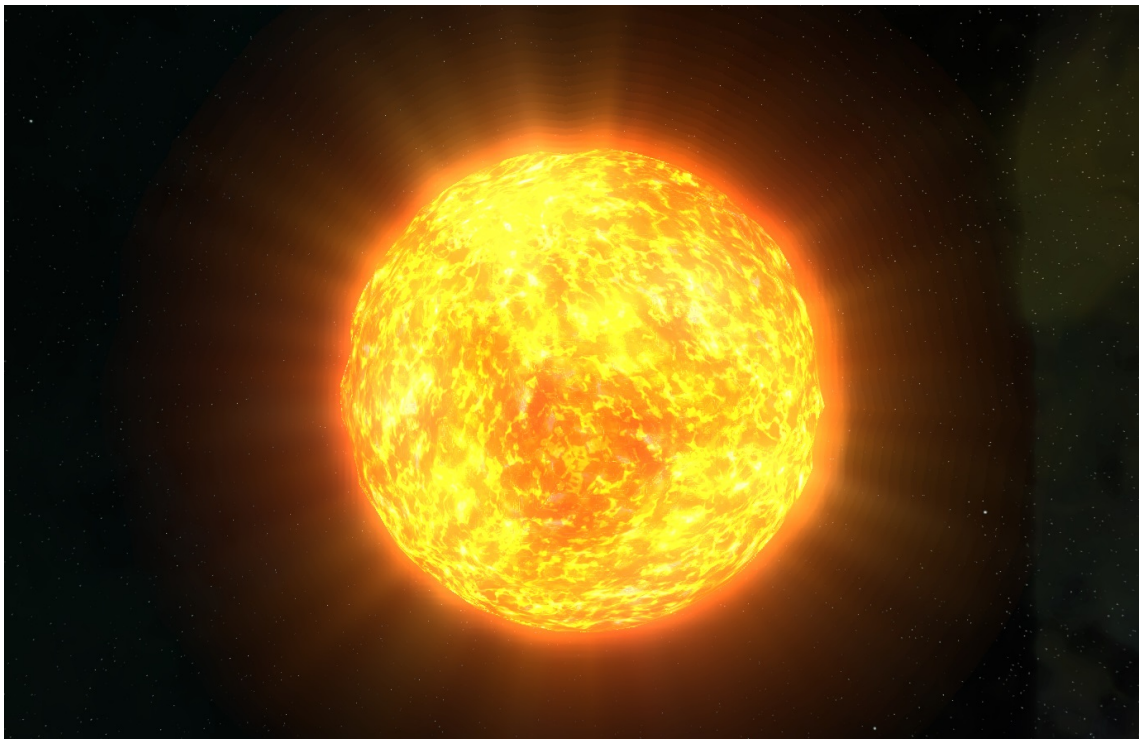
Demonstrační aplikace je naprogramována ve Visual Studiu 2015 a Visual Studiu 2017. Je použitá grafická knihovna OpenGL 4. Shader programy jsou naprogramovány ve výchozím jazyce GLSL. Jako verzovací systém kódu byl využit Git repozitář, jenž byl uložen na GitHubu a ovládán pomocí programu SourceTree.

4.3 Architektura demonstrační aplikace

Architektura demonstrační aplikace je založena na předchozí bakalářské práci, jejíž architektura byla založena převážně na architektuře Unity3D [32]. Ovšem místo statických tříd byl kladen důraz na použití instancí.

4.3.1 Vykreslování

Pro vykreslování je využit Defferred Shading [31], jenž nám umožní jednodušeji využít post efekty (anglicky post process effects) a efektivněji osvětlit scénu. Byly implementovány následující post efekty: Godrays, Bloom, Tonemapping. Post efekty jsou defaultně vypnuty, neboť jejich implementace je značně naivní. Byla využita také gama korekce barev. Pro potřeby vykreslení vody (transparentních objektů) se na konci vykreslovací smyčky provede normální Forward Rendering. Pokud je uživatel blízko povrchu planety, je prováděna jeho tesalace.



Obrázek 11: Ukázka post efektů Godrays a Bloom v demonstrační aplikaci.

4.4 Swap trick

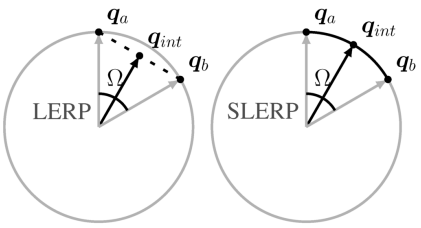
Jedním z problémů při tvorbě vykreslovací aplikace v reálném čase je, jak efektivně a rychle v multivláknové aplikaci upravovat seznam všech objektů pro vykreslení. V našem procedurálním světě se rychle generují nové segmenty a jiné stejnou rychlostí mizí. Pro mazání prvku ze seznamu byl využit takzvaný swap trick [30]: najdeme pozici prvku, na tuto pozici přesuneme prvek, jenž je na konci seznamu a snížíme interní počítadlo prvků. Swap z anglického výměna tedy implikuje, že vyměníme prvek, který chceme smazat s prvkem na konci seznamu. Nevýhodou toho přístupu je, že tento seznam neudržuje pořadí. To nám ovšem nevadí, protože při vykreslování se prvky tohoto seznamu stejně znovu seřadí podle vzdálenosti ke kameře.

4.5 Lineární a sférická interpolace

V průběhu vývoje demonstrační aplikace bylo zjištěno, že díky lineární interpolaci (anglicky nazývané Lerp) docházelo k nepřesnostem při výpočtu. Ze začátku se v algoritmech pro generování vrcholů pro segmenty využívala lineární interpolace, ovšem pokud tyto vrcholy poté promítneme na kulovou plochu, zjistíme že mezi nimi není stejná vzdálenost. To je problém, neboť pro plynulé a těžko rozpoznatelné zvětšení detailu chceme, aby detailnější segmenty měly pozice vrcholů na stejném místě jako rodičovský segment. Po výměně lineární interpolace za interpolaci sférickou (anglicky nazývané Slerp) byl tento problém vyřešen.

```
vec3 Lerp(vec3 start, vec3 end, float t) {
  return start * t + end * (1 - t);
}
vec3 Slerp(vec3 start, vec3 end, float t) {
  float startMagnitude = start.Length;
  float endMagnitude = end.Length;
  vec3 startNormalized = start / startMagnitude;
  vec3 endNormalized = end / endMagnitude;
  vec3 dot = startNormalized.Dot(endNormalized);
  dot = Clamp(dot, -1, 1);
  float theta = ACos(dot) * t;
  vec3 relativeVec = endNormalized - startNormalized * dot;
  return
    (startNormalized * Cos(theta)) + (relativeVec.Normalized() * Sin(theta))
    * Lerp(startMagnitude, endMagnitude, t);
}
```

Výpis 6: Ukázka funkcí pro lineární a sférickou interpolaci vektoru.



Obrázek 12: Ukázka rozdílu mezi lineární a sférickou interpolací [4].

[4] Převzato z: `http://www.mdpi.com/1424-8220/15/8/19302/htm`

4.6 Příprava dat pro vykreslení v samostatném vlákně

Velkým zpomalením v průběhu vykreslovací smyčky jsou také různé výpočty (například: Frustum Culling). Proto je ideální, když se vše vypočítá dopředu a poté jen zobrazí. Takto lze využít více vláken, kde jedno nebo více pracovních vláken počítá data, které se vykreslí v dalším snímku, zatímco jiné vykreslovací vlákno právě vykresluje poslední data v aktuálním snímku.

Po přesunu výpočtů do pracovních vláken bylo zaznamenáno významné zvýšení počtu snímků vykreslení za sekundu. I po této úpravě bylo zjištěno, že vykreslovací vlákno může čekat na pracovní vlákna. Po paralelizaci výpočtu v pracovních vláknech se počet snímků vykreslení za sekundu opět zvýšil.

4.6.1 Testování a výsledky

Tato optimalizace byla testována ve dvou pozicích kamery, neboť záleží na tom, kolik objektů chceme vykreslovat. Pozice kamery A je na povrchu planety a přes celou obrazovku je viditelný povrch planety. Pozice kamery B je pohled na planetu z vesmíru, kde planeta je vykreslená přibližně na jedne devítině obrazovky.

připravení dat pro vykreslení ve	PC1		PC2		PC3		PC4	
pozice kamery	A	B	A	B	A	B	A	B
vykreslovacím vlákně	90	154	76	130	4	15	143	406
pracovních vláknech	94	330	80	280	4	15	220	479

Tabulka 2: Vliv připravení dat pro vykreslení v samostatném vlákně na počet překreslení snímků za sekundu.

Z výsledku vidíme, že tato optimalizace je užitečná hlavně na konfiguracích s výkonnějšími grafickými kartami. Na těchto konfiguracích může nastat situace, ve které je grafická karta nevyužita kvůli toho, že vykreslovací vlákno právě připravuje data pro vykreslení, proto v těchto případech pomohlo data pro vykreslení připravit v samostatném vlákně.

4.7 Zeměpisné souřadnice

Pro potřeby mapování základní výškové textury nebo textury kontrolující biomy je potřeba mít texturovací souřadnice, tyto UV texturovací souřadnice jsou ekvivalentem zeměpisné šířky a délky. Směrový vektor ze středu planety se tedy převádí do zeměpisných souřadnic, které se pro tento účel využívají. Zeměpisné souřadnice se také využívají pro kolizi kamery s povrchem planety, kde potřebujeme kontrolovat a upravovat vzdálenost kamery od středu planety.

```
GeographicCoords ToGeographic(vec3 cartesian) {
  float altidute = cartesian.Length;
  if (altidute == 0) return GeographicCoords(0, 0, 0);
  return GeographicCoords(
```

```
    Math::Atan2(cartesian.Z, cartesian.X),
    Math::Asin(cartesian.Y / altidute),
    altidute
  );
}
vec3 ToCartesian(GeographicCoords geographic) {
  return vec3(
    Math::Cos(geographic.latitude) * Math::Cos(geographic.longitude) *
        geographic.altitude,
    Math::Sin(geographic.latitude) * geographic.altitude,
    Math::Cos(geographic.latitude) * Math::Sin(geographic.longitude) *
        geographic.altitude
  );
}

float Longitude01() {
    return longitude / (2 * Math::PI) + 0.5; // normalizace rozsahu do 0..1
}
float Latitude01() {
    return latitude / Math::PI + 0.5; // normalizace rozsahu do 0..1
}
```

Výpis 7: Ukázka funkcí pro převod do a z zeměpisných souřadnic.

4.8 Procedurální popis povrchu planety

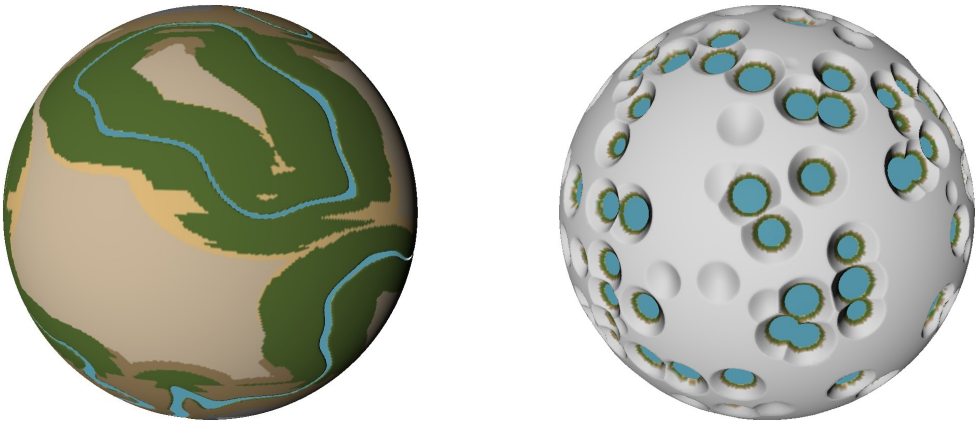
Vzdálenost povrchu od středu planety je popsána funkcí, jež obsahuje složení několika šumových funkcí. Do této metody vstupuje směrový normalizovaný vektor od středu planety. Tato funkce se také využívá pro získání finální pozice jakéhokoliv vrcholu na planetě, pokud známe jeho směr ze středu planety.

```
vec3 GetFinalPosition(vec3 direction) {
    vec3 dir = direction.Normalized();
    return dir * GetDistanceFromPlanetCenter(dir);
}
float GetDistanceFromPlanetCenter(vec3 dir) {
    float height = baseHeightMap.SampleAt(dir.ToGeographic().UV()).R();
    height *= this->planetRadius;
    height += GetPlanetHeight01(dir) * this->noiseHeightMultiplier;
    return height;
```

```
}
float GetPlanetHeight01(vec3 dir) {
    //continents
    float result = abs(PerlinNoise(dir*0.5, 5, 4));
    w = WorleyNoise(dir*2);
    result += (w.x - w.y) * 2;
    //oceans
    result -= abs(PerlinNoise(dir*2.2, 4, 4));
    //big rivers
    x = PerlinNoise(dir * 3, 3, 2);
    result += -exp(-pow(x*55,2)) * 0.2;
    //craters
    w = WorleyNoise(dir);
    result += smoothstep(0.0, 0.1, w.x);
    //small detail
    float p = PerlinNoise(dir*10, 5, 10) * 100;
    result += terrace(p, 0.3)*0.005;
    result += p*0.005;
    return result;
}
float PerlinNoise(vec3 position, int octaves, float persistance) {
    float result = 0;
    float amplitude = 1;
    for (int i = 0; i < octaves; i++) {
        result += PerlinNoise(position) * amplitude;
        position *= persistance;
        amplitude /= persistance;
    }
    return result;
}
```

Výpis 8: Kód pro procedurální popis povrchu planety.

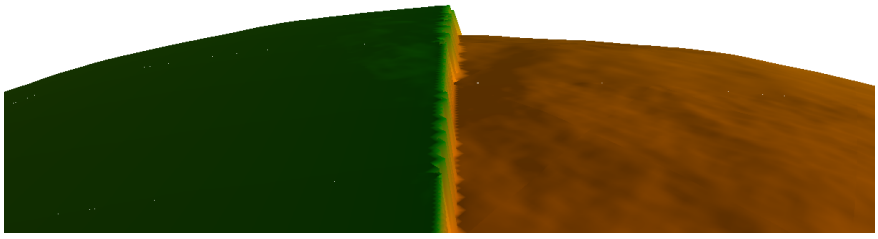
Pro procedurální popis povrchu planety se využívají pouze dvě šumové funkce (Perlinův šum a Worleyho šum). Pomocí jejich rozdílných kombinací a použití se dosahuje tíženého výsledku. Například v předcházejícím kódu lze vidět, že řeky jsou pouze upraveným výsledkem Perlinova šumu.



Obrázek 13: Ukázka využití šumových funkcí pro tvorbu řek a kráterů.

4.8.1 Dvojrozměrně a trojrozměrně šumové funkce

Zpočátku tvorby demonstrační aplikace byly použity dvojrozměrné šumové funkce, do nichž vstupovala zeměpisná šířka a délka ze zeměpisných souřadnic. Ovšem díky náhlému skoku souřadnic na poledníku, se na poledníku objevil zlom. Tento zlom byl odstraněn po použití trojrozměrných šumových funkcí, do nichž vstupoval trojrozměrný směrový vektor ze středu planety.



Obrázek 14: Ukázka zlomu na poledníku kvůli použití dvojrozměrných šumových funkcí.

4.9 Tvar kořenových segmentů

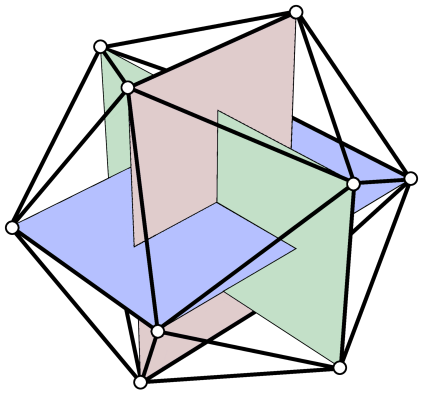
V naší práci tedy mají rozsahy kořenových segmentů tvar rovnostranných trojúhelníků složených z dvacetistěnů. Tento tvar dosahuje pro nás nejlepších vlastností. Ovšem generování trojúhelníkové sítě, která vyplňuje náš rozsah segmentu ve tvaru trojúhelníku je složitější, než generování do tvaru čtverce.

Následující kód je použit pro výpočet rozsahu dvaceti kořenových segmentů, jedná se o algoritmus pro konstrukci dvacetistěnu.

```
float t = (1 + sqrt(5.0)) / 2.0 * planetRadius / 2.0;
float d = planetRadius / 2.0;

vec3 v0 = vec3(-d, t, 0); vec3 v1 = vec3(d, t, 0);
vec3 v2 = vec3(-d, -t, 0); vec3 v3 = vec3(d, -t, 0);
```

```
vec3 v4 = vec3(0, -d, t); vec3 v5 = vec3(0, d, t);
vec3 v6 = vec3(0, -d, -t); vec3 v7 = vec3(0, d, -t);
vec3 v8 = vec3(t, 0, -d); vec3 v9 = vec3(t, 0, d);
vec3 v10 = vec3(-t, 0, -d); vec3 v11 = vec3(-t, 0, d);

AddRootSegment(v0, v11, v5); AddRootSegment(v0, v5, v1);
AddRootSegment(v0, v1, v7);  AddRootSegment(v0, v7, v10);
AddRootSegment(v0, v10, v11); AddRootSegment(v1, v5, v9);
AddRootSegment(v5, v11, v4); AddRootSegment(v11, v10, v2);
AddRootSegment(v10, v7, v6); AddRootSegment(v7, v1, v8);
AddRootSegment(v3, v9, v4);  AddRootSegment(v3, v4, v2);
AddRootSegment(v3, v2, v6);  AddRootSegment(v3, v6, v8);
AddRootSegment(v3, v8, v9);  AddRootSegment(v4, v9, v5);
AddRootSegment(v2, v4, v11); AddRootSegment(v6, v2, v10);
AddRootSegment(v8, v6, v7);  AddRootSegment(v9, v8, v1);
```

Výpis 9: Kód pro výpočet kořenových segmentů.



Obrázek 15: Vizualizace konstrukce dvacetistěnu [5].

4.10 Selektivní zvětšování detailu planety

Pokud chceme zvětšit detail jednoho segmentu, rozdělíme jeho rozsah na další čtyři stejně velké segmenty a původní segment skryjeme. Nás algoritmus výběru segmentů pro rozdělení se snaží udržet velikost každého viditelného segmentu na obrazovce menší, než předem určená reálná konstanta. Metoda pro výpočet velikosti segmentu na obrazovce může vypadat například následovně.

[5] Převzato z: `http://www.inventortales.com/2012/10/all-for-fun-creating-20-sided_26.html`

```
class Segment {
  float GetSizeOnScreen() {
    float distanceToCamera = this->WorldPosition.Distance(Camera->Position);
    float radiusWorldSpace = this->BoundingSphere.Radius;
    float radiusCameraSpace = radiusWorldSpace * Math::Cot(Camera->FieldOfView
        / 2) / distanceToCamera;
    return radiusCameraSpace;
  }
}
```

Výpis 10: Kód pro výpočet přibližné velikosti segmentu na obrazovce.

Tuto metodu lze dále zaobalit a modifikovat různými koeficienty, těmito koeficienty může být například úhel segmentu ke kameře, nebo zda byl segment viditelný minulý snímek.

```
class Segment {
  float GetGenerationWeight() {
    float w = this->GetSizeOnScreen();
    float directionToCamera = this->WorldPosition.Towards(Camera->Position).
        Normalized();
    w *= 1 + Math::Clamp01(this->Range.Normal.Dot(directionToCamera));
    if(this->Renderer->WasVisibleLastFrame == false) w *= 0.3;
    return w;
  }
}
```

Výpis 11: Kód pro výpočet váhy segmentu pro generování trojúhelníkové sítě.

V demonstrační aplikaci využíváme pouze modifikaci váhy, pokud byl segment viditelný minulý snímek.

Samotné logické rozdělení segmentu na jeho potomky (metoda CreateChildren) se vykoná ihned při počítání vah, neboť toto rozdělení je rychlé. Ovšem o několik řádů pomalejší je generování trojúhelníkové sítě pro segmenty. Proto se toto logické rozdělení segmentů a tvorba seřazeného listu segmentů pro generování počítá v samostatném vlákně. Samotné generování trojúhelníkové sítě segmentu se poté vykonává na konci každého vykreslení snímku ve vykreslovacím vlákně. Toto vykreslovací vlákno konzumuje náš váhově seřazený list segmentů pro vygenerování.

```
class Planet {

  WeightedSegmentsList* toGenerateList = new WeightedSegmentsList();
```

```
  void TrySubdivide() {
    foreach (Segment* rootSegment in rootSegments) {
      if (rootSegment->GenerationBegan == false)
        toGenerateList->Add(rootSegment, Constants::floatMaxValue);
      else
        TryAddSegment(rootSegment, 0);
    }
  }

  void TryAddSegment(Segment* segment, int recursionDepth) {
    float weight = segment->GetGenerationWeight();

    if (segment->GenerationBegan == false)
      toGenerateList->Add(segment, weight);

    if (recursionDepth < maxRecurisonDepth && weight > weightNeededToSubdivide)
        {
      segment->EnsureChildrenAreCreated();

      foreach (Segment* child in segment->Children)
        TryAddSegment(child, recursionDepth + 1);

      if(segment->AllChildrenFinishedGeneration())
        segment->Hide();
      else
        segment->ShowAndHideDescendants();

      return;
    }
    if (segment->IsGenerationDone)
      segment->ShowAndHideDescendants();
  }

}
```

Výpis 12: Kód pro výpočet váhově seřazeného seznamu segmentů pro vygenerování trojúhelníkových sítí.

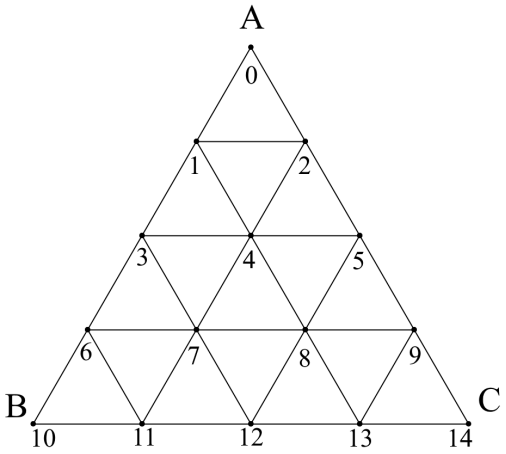
4.11 Generování trojúhelníkové sítě

Pokud chceme, aby segment byl viditelný, musíme nejdříve vygenerovat jeho trojúhelníkovou síť. Nejjednodušším algoritmem pro generování trojúhelníkové sítě jednoho segmentu je rekurzivní metoda, která prochází všechny existující trojúhelníky a vždy je rozdělí na další čtyři. Tato metoda ovšem zanechá velké množství duplicitních vrcholů, které se musí následně odstranit. V naší práci jsme tedy vymysleli jednodušší metodu, která vygeneruje celou trojúhelníkovou síť segmentu v jednom průchodu a nezanechá žádné duplicitní vrcholy. Tato optimalizace se ovšem ukázala být zbytečná, neboť tvorba těchto polí (indicie a vrcholy) proběhne v demonstrační aplikaci pouze jednou a poté se pouze kopíruje.

```
allTriangles->Clear();
allTriangles->Add(segment->range);
for(int i=0; i<numberOfIterations; i++) {
  newAllTriangles->Clear();
  foreach(Triangle triangle in allTriangles) {
    vec3 a = triangle.a;
    vec3 b = triangle.b;
    vec3 c = triangle.c;
    vec3 ab = (a + b) / 2;
    vec3 ac = (a + c) / 2;
    vec3 bc = (b + c) / 2;
    newAllTriangles->AddTriangle(a, ab, ac);
    newAllTriangles->AddTriangle(b, bc, ab);
    newAllTriangles->AddTriangle(c, ac, ac);
    newAllTriangles->AddTriangle(ac, bc, ab);
  }
  allTriangles = newAllTriangles;
}
RemoveDuplicitVertices();
```

Výpis 13: Kód pro rekurzivní zjemňování trojúhelníkové sítě, jež zanechá duplicitní vrcholy.

Nevýhodou jednoduchého rekurzivního generování trojúhelníkové sítě je také nevhodné pořadí, v jakém se vrcholy nachází. Pro potřeby dalších algoritmů by bylo lepší, aby pořadí vrcholů bylo pevně dané a jednoduše odvoditelné. Naše trojúhelníková sít segmentu má tedy následující strukturu.



Obrázek 16: Struktura trojúhelníkové sítě segmentu o 15 vrcholech.

Z této struktury lze potom odvodit následující vztahy, jež se dále využívají například pro vytvoření sukní. V následujících vztazích: $a, b, c \in \mathbb{N}$ jsou indexy vrcholů A, B, C, $e \in \mathbb{N}$ je počet vrcholů na hraně a $t \in \mathbb{N}$ je celkový počet vrcholů v trojúhelníkové síti.

$$\begin{aligned} e &= 5 \\ t &= \frac{(e-1)e}{2} + e \\ \\ a &= 0 \\ b &= \frac{(e-1)e}{2} \\ c &= b + (e-1) \end{aligned} \tag{1}$$

Kód, jenž nám vygeneruje indicie naší trojúhelníkové sítě v žádané struktuře, vypadá takto:

```
int start = 0;
int nextStart = 1;
indicies->Add(0, 1, 2); // first triangle
int between = 0; // number of vertices between start and end
for (int y = 1; y < e - 1; y++)
{
  start = nextStart;
  nextStart = start + between + 2;
  for (int x = 0; x <= between + 1; x++)
  {
    indicies->Add(start + x, nextStart + x, nextStart + x + 1);
    if (x <= between) // not a last triangle in line
```

```
            indicies->Add(start + x, nextStart + x + 1, start + x + 1);
        }
        between++;
    }
```

Výpis 14: Optimalizovaný kód pro generování indicií pro trojúhelníkovou síť pro libovolný počet vrcholů na hraně.

Nyní potřebujeme projít všechny vrcholy a vypočítat jejich finální pozice. Zde se využívá výše zmíněná funkce *GetFinalPosition*, do které vstupuje směrový vektor od středu planety a která nám vrátí finální pozici vrcholu na povrchu planety. V tomto kódu je použita také zmíněná sférická interpolace.

```
int between = 0; // number of vertices between start and end
for (uint y = 1; y < e; y++) {
    float percent = y / (e - 1);
    vec3 start = Math::Slerp(this->range.a, this->range.b, percent);
    vec3 end = Math::Slerp(this->range.a, this->range.c, percent);
    vertices->Add(planet->GetFinalPosition(start));
    if (between > 0) {
        for (uint x = 1; x <= between; x++) {
            vec3 v = Math::Slerp(start, end, x / (between + 1));
            vertices->Add(planet->GetFinalPosition(v));
        }
    }
    vertices->Add(planet->GetFinalPosition(end));
    between++;
}
```

Výpis 15: Kód pro vygenerování pozic (výšek) vrcholů pro trojúhelníkovou síť segmentů.

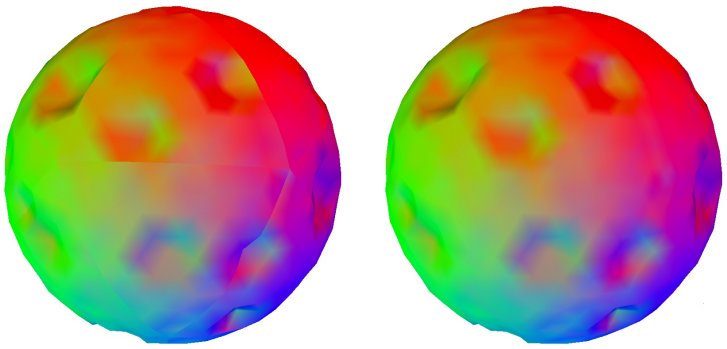
4.12 Plynule navazující normály segmentů a zakrytí prasklin

Normály segmentů dopočítáváme z pozic vrcholů, pro každý segment spočítáme jeho normály pomocí jeho vrcholů. Výsledkem bylo, že normály na hranici dvou segmentů nenavazovaly. Byly vyzkoušeny dvě metody pro vyhlazení normál na hranách segmentů.

1. Normály vrcholů, které jsou na stejném místě, ale v sousedících segmentech, zprůměrovat poté, až se segmenty vygenerují. Tato metoda je nejjednodušší a má nejlepší vizuální výsledky, neboť garantuje identické normály na hranách. Ovšem u této metody je nevýhoda, že je potřeba jí počítat pokaždé, když se sousedící segmenty změní.

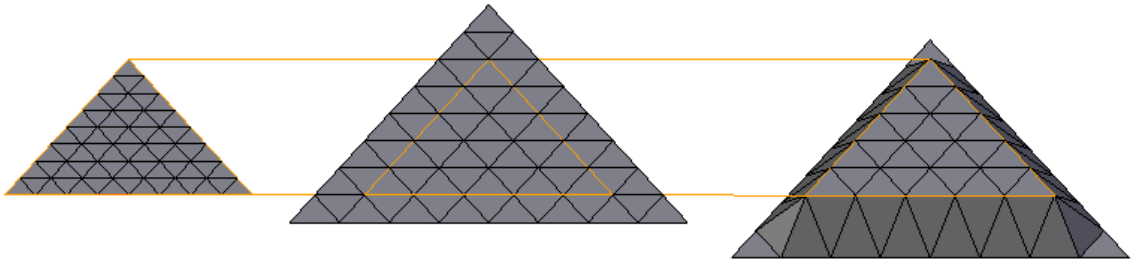
2. Zvětšení segmentu a odříznutí přečnívající části. Tím se zajistí návaznost normál na hranách. U této metody můžeme přečnívající části buď odmazat, nebo je můžeme použít na tvorbu sukní. Nevýhodou je, že sousedící segmenty mohou být z rozdílných generací, což znamená, že mohou nastat případy, kdy normály na hranách nenavazují dokonale.

První metoda dosahovala podle očekávání nejpřesnějších výsledků, ovšem musela se počítat pokaždé, když se změnily sousedící segmenty. Proto nakonec byla použita a implementována metoda 2.



Obrázek 17: Základní segmenty planety před a po zprůměrování normál na hranách, normály jsou vizualizovány barvou.

Pro účely druhé metody potřebujeme vypočítat, o kolik chceme rozsah segmentu zvětšit. Naše trojúhelníková síť má strukturu definovanou v předcházejících kapitolách (obrázek [16] a vztahy [1]). Segment tedy zvětšíme, ideálně přesně o vzdálenost jednoho vrcholu, což nám umožní spočítat normály za opravdovým okrajem rozsahu segmentu. Přečnívající část poté posuneme směrem do středu planety. Tato posunutá přečnívající část nám vytvoří sukně, které zakryjí potenciální praskliny.



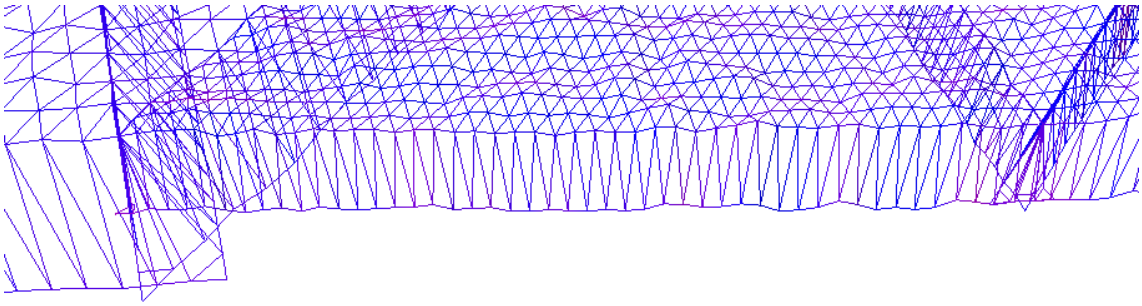
Obrázek 18: Ukázka zvětšení segmentu přesně o vzdálenost dvou vrcholů a následné využití přečnívajících částí pro sukně.

Následující vztahy definují, jak modifikovat rozsah segmentu (pozice rohů A, B, C), aby se zvětšil přesně o vzdálenost jednoho vrcholu na trojúhelníkové síti. V následujících vztazích platí: $r, m \in \mathbb{R}$ a Z, A, B, C jsou souřadnice $[x, y, z]$.

$$r = \frac{1}{e-3}$$
$$m = 1 + \sqrt{(2r)^2 - r^2}$$

$$Z = \frac{A+B+C}{3}$$

$$A = (A - Z)m + Z;$$
$$B = (B - Z)m + Z;$$
$$C = (C - Z)m + Z;$$



Obrázek 19: Ukázka sukní v demonstrační aplikaci.

4.13 Přesunutí výpočtů na grafickou kartu

V průběhu vývoje demonstrační aplikace bylo zjištěno, že výpočet pozic vrcholů trojúhelníkových sítí pro segmenty na hlavním procesoru trvá dlouhou dobu, proto bylo co nejvíce výpočtů přesunuto na grafickou kartu. Pro výpočty na grafické kartě byl využit OpenGL Computer Shader [33]. Paralelizace byla v tomto případě přímočará, neboť pro každý vrchol bylo nutno vypočítat řadu šumových funkcí (a čtení textur), jež jsou nezávislé na ostatních vrcholech.

Demonstrační aplikace využívá pouze jeden OpenGL kontext. To znamená, že všechnu práci z OpenGL musíme synchronizovat, z toho důvodu jakákoliv práce v OpenGL včetně compute shaderů musí probíhat ve stejném vlákně. Všechna práce z OpenGL tedy probíhá ve vykreslovacím vlákně. To znamená, že doba, po kterou se segmenty počítají na grafické kartě, nám snižuje počet překreslení snímku za sekundu. Z grafu porovnání časů na hlavním procesoru a grafické kartě (obrázek [21]) je viditelné, že na konfiguraci PC2 při generování trojúhelníkové sítě segmenty byla grafická karta zatížena až na 0.142 sekund. To znamená, že po dobu 0.142 sekund jsme nemohli znovu vykreslit snímek, což nám snížilo počet překreslení snímků za sekundu až

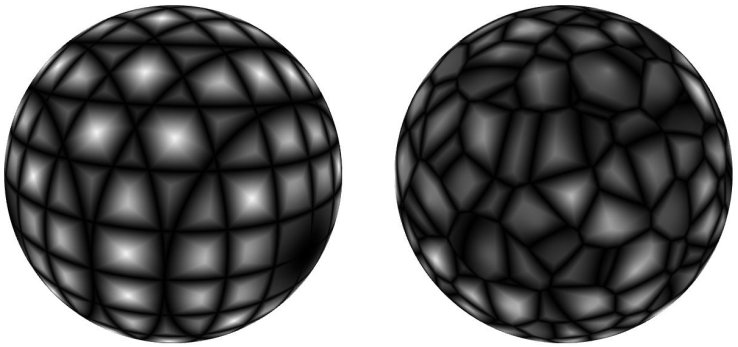
na 7 (1/0.142). Je tedy nutné snížit počet vrcholů na segmentu nebo výpočet rozdělit do více kroků.

Pokud bychom snížili počet vrcholů na segmentu, potřebovali bychom více segmentů, abychom dosáhli stejného detailu. To znamená, že budeme mít větší počet objektů pro vykreslení, což nám může opět negativně ohrozit počet vykreslení snímků za sekundu.

Generování trojúhelníkové sítě segmentu tedy bylo rozděleno do několika samostatných kroků:

1. Vytvoření trojúhelníkové sítě a vykreslovací komponenty.
2. Přesun trojúhelníkové sítě na grafickou kartu.
3. Vygenerování výšek trojúhelníkové sítě na grafické kartě.
4. Kopírování trojúhelníkové sítě z grafické karty do hlavní paměti počítače.
5. Výpočet obalového kvádru trojúhelníkové sítě (pro Frustum Culling).
6. Výpočet normál trojúhelníkové sítě na grafické kartě.
7. Vytvoření sukní.

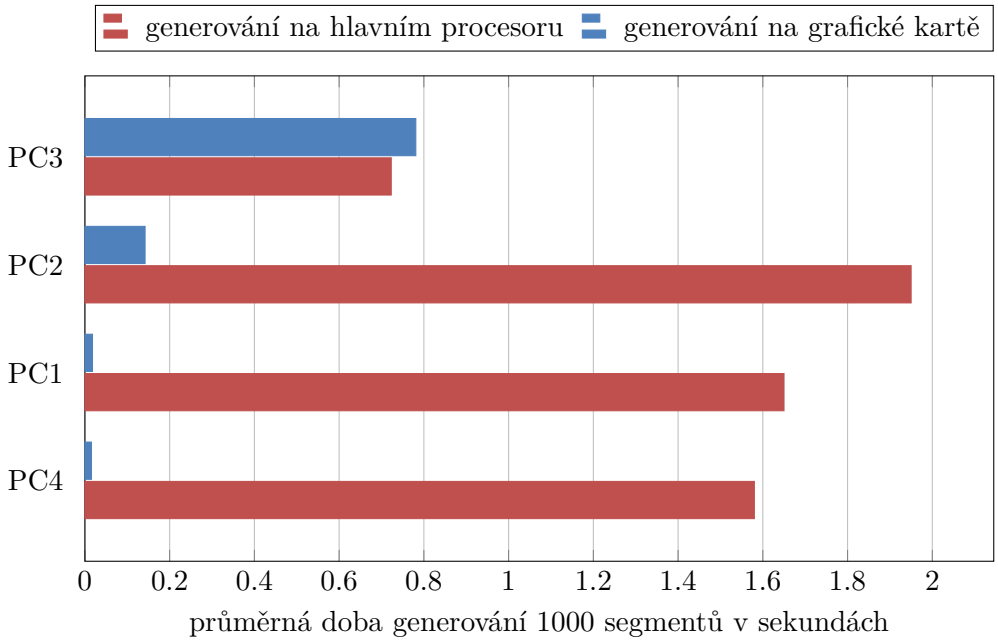
Demonstrační aplikace má díky tomuto rozdělení více příležitostí pozastavit výpočet, pokud detekuje, že spuštění dalšího kroku by překročilo náš limit 60 překreslení snímků za sekundu. Při tomto řešení není nutné snižovat počet vrcholů na segmentu a dosáhne se stabilnějšího počtu překreslení snímků za sekundu.



Obrázek 20: Ukázka Worleyho šumu D1-D0 na grafické kartě, nalevo jitter 0, napravo jitter 1.

4.13.1 Testování a výsledky

Z tabulky porovnání časů na hlavním procesoru a grafické kartě (obrázek [21] jde vidět, že výpočet převážně na grafické kartě dosahuje v závislosti na typu grafické karty až stonásobné zrychlení.



Obrázek 21: Změna průměrné doby generování 1000 segmentů po přesunu výpočtů na grafickou kartu.

Pokud tyto časy detailněji rozepíšeme (tabulka [3]), lze si všimnout, že zejména na novějších grafických kartách trvá krok kopírování trojúhelníkové sítě z grafické karty do hlavní paměti nejdelší dobu.

	PC1	PC3	PC4
celkem	17.90ms	781.0ms	15,50ms
vytvoření trojúhelníkové sítě a vykreslovací komponenty	263.4us	140.7us	114,8us
přesun trojúhelníkové sítě na grafickou kartu	2.621ms	1.848ms	1,774ms
vygenerování výšek trojúhelníkové sítě na grafické kartě	73.43us	408.7ms	45,04us
kopírování trojúhelníkové sítě z grafické karty	14.43ms	370.1ms	13,38ms
výpočet obalového kvádru trojúhelníkové sítě	90.76us	45.17us	45,01us
výpočet normál trojúhelníkové sítě na grafické kartě	166.7us	70.67us	71,66us
posunutí vrcholů pro vytvoření sukní	32.37us	4.250us	7,805us
přesun upravené trojúhelníkové sítě na grafickou kartu	189.6us	39.38us	45,65us
ukončení generování	32.08us	17.31us	14,22us

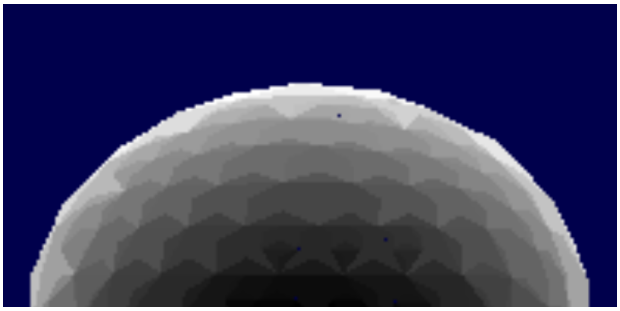
Tabulka 3: Změna průměrné doby generování 1000 segmentů po přesunu výpočtů na grafickou kartu, rozděleno na jednotlivé kroky.

Trojúhelníkové sítě se kopírují z grafické karty do hlavní paměti pro účely kolize kamery. Pravě toto kopírování trvá nejdelší dobu. Teoreticky lze OpenGL informovat, že tuto trojúhelníkovou síť budeme často kopírovat (pomocí OpenGL buffer hint) [33]. Po informování OpenGL

se ukázalo, že doba kopírování trojúhelníkové sítě z grafické karty do hlavní paměti opravdu trvala kratší dobu, ovšem ostatní kroky, které pracují s grafickou kartou trvaly déle.

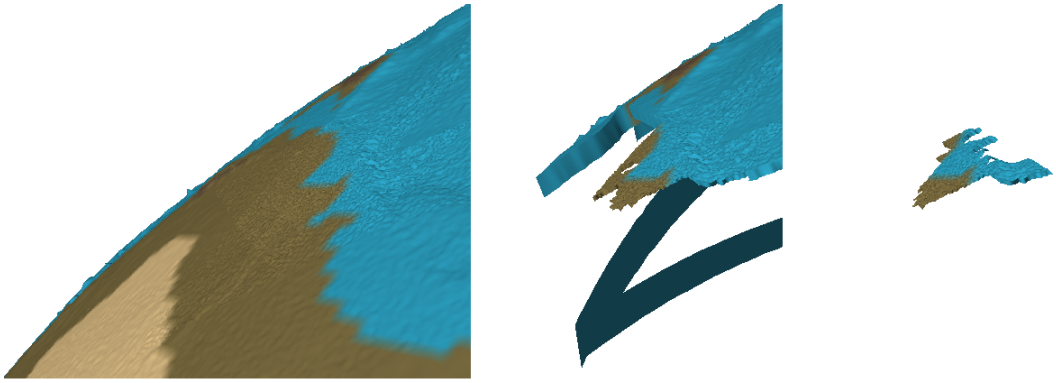
4.14 Nevykreslování skrytých ploch

I přes použíti optimalizačního algoritmu Frustum Culling se pořád vykresluje mnoho zbytečných segmentů, což není ideální. Na tento problém byl implementován softwarový z bufer rasterizér, neboli softwarový rasterizér hloubky scény. Pro každý segment se vytvoří Occluder sestávající se ze čtyř trojúhelníků. Tyto trojúhelníky se poté razterizují do softwarového hloubkového buferu o velikosti 200 na 100 bodů. Tato malá velikost softwarového z buferu je k zjištění viditelnosti objektů dostačující [30]. Při rasterizaci jakéhokoliv trojúhelníku se pro optimalizaci zapisuje konstantní hloubka definovaná nejvzdálenějším vrcholem trojúhelníku. Tento algoritmus je konzervativní, proto si můžeme dovolit zapisovat hloubku nejvzdálenějšího vrcholu. Konzervativní v tomto kontextu znamená, že nám nevadí když algoritmus vypočítá, že některý objekt je viditelný i když ve skutečnosti je skryt. Vadilo by nám, kdyby algoritmus řekl, že objekt je skryt, ale ve skutečnosti by byl viditelný.



Obrázek 22: Vizualizace softwarového z buferu.

Rasterizovali jsme tedy occludery všech potenciálních objektů pro vykreslení, jež prošly frustum cullingem. Nyní je potřeba pomocí právě softwarově vykresleného z buferu zjistit, které z těchto objektů budou pravděpodobně viditelné. Projdeme tedy všechny potenciální objekty a převedeme jejich obalové tělesa do kvádrů v kamerovém prostoru. Jelikož je algoritmus konzervativní, stačí otestovat, zdali alespoň jeden bod v softwarovém z buferu má hloubku větší, než nejmenší hloubka kvádru v kamerovém prostoru. Pokud ano, objekt je pravděpodobně viditelný.



Obrázek 23: Ukázka nevykreslování skrytých ploch v demonstrašní aplikaci (zleva: vypnutí nevykreslování skrytých ploch, pouze Frustum Culling, Frustum Culling a softwarový z bufer).

```
rasterizer->Clear();
foreach(Renderable* renderable in possibleRenderables) {
    Bounds bounds = renderable->GetFloatingOriginSpaceBounds(camera);
    if (cameraFrustum->SphereVsFrustum(bounds.Center, bounds.Extents.Length) &&
        cameraFrustum->VolumeVsFrustum(bounds)) {
      passedFrustumCulling.Add(renderable);
      rasterizer->AddTriangles(renderable->GetOccluderTriangles(camera));
    }
}
foreach(Renderable* renderable in passedFrustumCulling) {
    CameraSpaceBounds bounds = renderable->GetCameraSpaceBounds(camera);
    if (rasterizer->AreBoundsVisible(bounds))
      toRender->Add(renderable);
}
```

Výpis 16: Zjednodušená smyčka pro připravení dat pro vykresleni z frustum cullingem a softwarovým z bufer rasterizérem.

4.15 Optimalizace triplanárního mapováni

Pro potřeby texturování povrchu planety využíváme několika textur v několika měřítkách, kde všechny tyto textury jsou namapovány pomocí triplanární mapování. Zjistili jsme, že toto kombinování mnoha triplanárních mapování s rozdílnými měřítky má viditelný vliv na počet vykreslených snímků za sekundu. Každé triplanární mapování čte 3 textury, pokud tedy využíváme 6 triplanárních mapování, náš vykreslovací fragment shader musí číst $6 * 3 = 18$ textur. Tento počet lze snížit, pokud triplanarní mapování optimalizujeme tak, aby se textur četlo co nejméně a aby se četly jen ty textury, které jsou nezbytně potřeba.

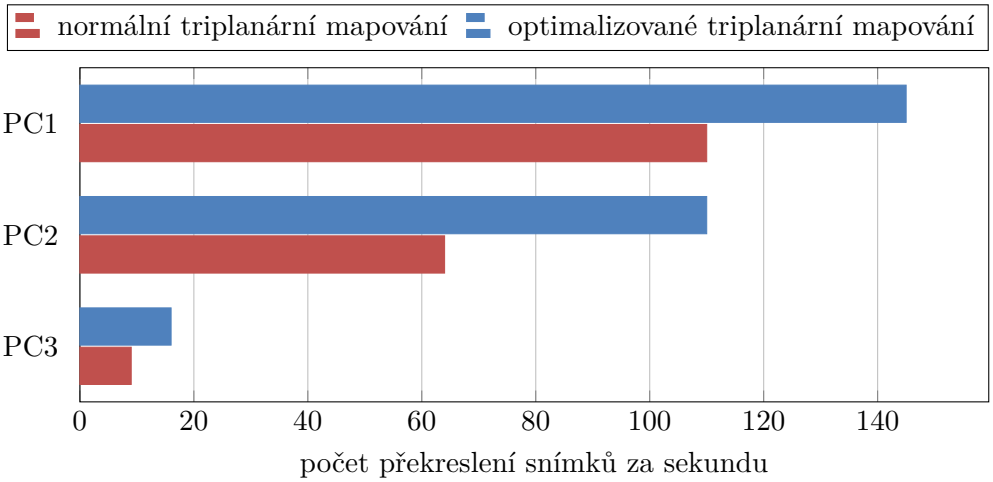
```
vec3 triplanar(sampler2D texture, vec3 position, vec3 normal) {

    vec3 blendWeights = pow(abs(normal), vec3(20));
    blendWeights /= blendWeights.x + blendWeights.y + blendWeights.z;

    vec3 result;

    const float threshold = 0.05;

    vec3 finalWights = vec3(0);

    if(blendWeights.x > threshold) finalWights.x = blendWeights.x;
    if(blendWeights.y > threshold) finalWights.y = blendWeights.y;
    if(blendWeights.z > threshold) finalWights.z = blendWeights.z;

    finalWights /= finalWights.x + finalWights.y + finalWights.z;

    if(finalWights.x > 0) result += finalWights.x * texture2D(tex, position.yz
        * scale).xyz;
    if(finalWights.y > 0) result += finalWights.y * texture2D(tex, position.zx
        * scale).xyz;
    if(finalWights.z > 0) result += finalWights.z * texture2D(tex, position.xy
        * scale).xyz;

    return result;
}
```

Výpis 17: Optimalizovaný shader pro triplanární texturování.

Po implementaci a vyzkoušení optimalizovaného triplanárního mapování se na hranicích dvou rozdílných měřítek textur objevil artefakt. Bylo zjištěno, že artefakt je způsoben mip mapama. Mip mapy při svém výpočtu využívají derivaci 2x2 čtverce soudících pixelu. Implementace mip map tedy předpokládá, že na jednom řádku shaderu se přistupuje ke stejné textuře v přibližně stejném měřítku, pokud tomu tak není, mip mapy vypočítají špatnou derivaci a tím pádem způsobí nechtěný artefakt [33]. Bylo tedy potřeba upravit shader tak, aby sousedící pixely četly texturu ve stejném měřítku.

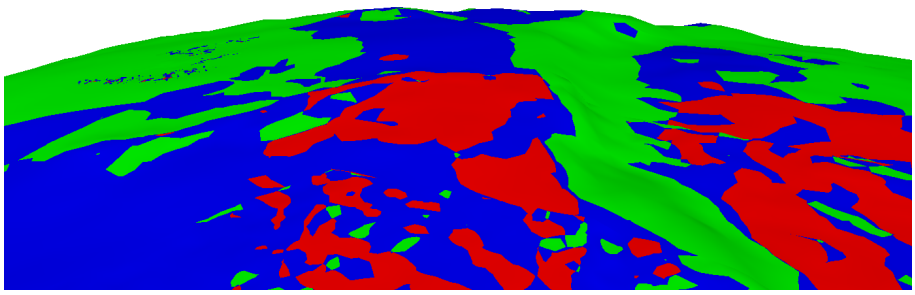
4.15.1 Testování a výsledky

Pro testování byla kamera umístěna tak, aby na celé obrazovce byl vidět pouze povrch planety, což znamená, že pro každý pixel se provádí triplanární mapování planety. Rozlišení vykreslovacího okna bylo 1400 na 900 pixelů. Je možné, že na jiných místech planety mohou být výsledky trochu odlišné.



Obrázek 24: Vliv optimalizovaného triplanárního mapování na počet překreslení snímků za sekundu.

Je viditelné, že optimalizace triplanárního mapování má značný vliv na počet překreslení snímků za sekundu, zejména u méně výkonných grafických karet lze pozorovat téměř 80% zrychlení.



Obrázek 25: Vizualizace počtu čtení textur při optimalizovaném triplanárním mapováni (zelená barva jedno čtení, modra barva dvě čtení, červená barva tři čtení textur).

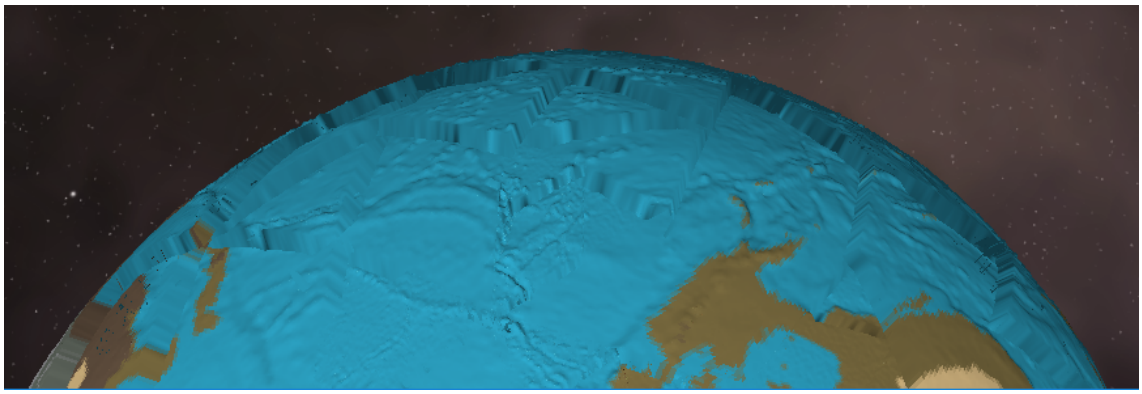
4.16 Z Fighting

Po použití sukní bylo zjištěno, že díky jevu zvaném Z Fighting, se sukně někdy vykreslovaly nad našimi segmenty [34]. Hodnoty v OpenGL hloubkovém buferu (anglicky Depth Buffer) se skládají

z hodnot (F_{depth}), jež reprezentují rozsah od z_{near} k z_{far}. Zmíněné z_{near} a z_{far}, jsou hodnoty v nastavení kamery, jež definují vzdálenosti přední a zadní ořezové roviny kamery.

$$F_{depth} = \frac{z - z_{near}}{z_{far} - z_{near}}$$

Přesnost hloubkového OpenGL buferu je tedy závislá na poměru z_{near}/z_{far}, čím je tento poměr menší, tím je menší i přesnost hloubkového buferu. Naším problémem je, že potřebujeme z_{near} menší než dva metry, neboť kamera se může pohybovat po povrchu planety až ve vzdálenosti dvou metrů. A dále potřebujeme co největší z_{far}, abychom viděli celou planetu.



Obrázek 26: Ukázka Z Fighting v demonstrační aplikaci, sukně které jsou pod segmentem, se vykreslují nad segmenty.

Tento problém lze vyřešit například těmito způsoby:

- Vykreslit scénu pomocí více kamer, jejichž jednotlivé z_{near} a z_{far} je kaskádou originálního z_{near} a z_{far}.
- Implementovat vlastní z bufer, jenž bude mít větší přesnost.
- Dynamické z_{near} a z_{far}: Pokud jsme daleko od planety, nepotřebuje male z_{near}. A pokud jsme blízko a na planetě, nepotřebujeme velké z_{far}.

Pro nás nejjednodušším řešením je dynamické z_{near} a z_{far}. Následující hodnoty se ukázaly být pro naše účely dostačující.

```
float s = Math::SmoothStep(1, 30000, cameraDistanceToPlanetSurface);
camera->ZNear = 1000 * s + 0.5;
camera->ZFar = 5000000 * s + 100000;
```

Výpis 18: Kód pro výpočet dynamického nastavení z_{near} a z_{far} kamery.

4.17 Identifikační číslo segmentů

V jistých případech, například pro generování objektů na povrchu planety, můžeme potřebovat deterministické seed. Jako deterministické seed může sloužit například ID (identifikační číslo) segmentů. Kořenové segmenty mají ID od 0 až 19. ID dalších dceřinných segmentů lze poté dopočítat pomocí bitového posunutí ID současného segmentu o dva bity a přičtení pořadí dceřiného segmentu.

```
class Segment {
    void CreateChildren() {
        vec3 a = this->range.a;
        vec3 b = this->range.b;
        vec3 c = this->range.c;
        vec3 ab = ((a + b)/2).Normalized() * this->planet->radius;
        vec3 ac = ((a + c)/2).Normalized() * this->planet->radius;
        vec3 bc = ((b + c)/2).Normalized() * this->planet->radius;

        AddChild(a, ab, ac, this->id << 2 | 0);
        AddChild(ab, b, bc, this->id << 2 | 1);
        AddChild(ac, bc, c, this->id << 2 | 2);
        AddChild(ab, bc, ac, this->id << 2 | 3);
    }
    void AddChild(vec3 a, vec3 b, vec3 c, ulong id) { ... }
}
```

Výpis 19: Kód pro rozdělení segmentu na jeho potomky.

4.18 Nepřesnost desetiného čísla float

OpenGL a knihovny, jež podporují práci z OpenGL, pracují převážně s 32 a 64 bitovými desetinnými čísly podle standartu IEEE 754 [33]. Desetinná čísla podle tohoto standartu se nazývají float a double. Tyto čísla se používají ve všech fázích vykreslovacího řetězce a nesou sebou řadu výhod i nevýhod. Jednou z nevýhod je, že čím větší hodnota je uložená v tomto desetinném čísle, tím se zmenšuje jeho přesnost.

V naší aplikaci můžeme kamerou libovolně pohybovat, čím vzdálenější je kamera od středu scény $[0, 0, 0]$, tím se zvětšuje riziko artefaktů spojených s nepřesností desetinného čísla. Jedním z takovýchto artefaktů je například nepřirozený (skokový) pohyb kamery, protože při velkých hodnotách již nelze přičíst náš relativně malý pohybový vektor.

```
1000000   + 1 = 1000001
10000000  + 1 = 10000001
100000000 + 1 = 100000000
```

Výpis 20: Ukázka operací nad float.

Vidíme, že k 100000000 jíž nelze přičíst 1. Znamená to tedy, že pokud bychom chtěli simulovat sluneční soustavu, například pohyb planety Země ve vzdálenosti 149 milionů kilometrů od slunce, nebylo by již možné v této vzdálenosti kamerou plynule pohybovat.

Tento problém musíme vyřešit jak na hlavním procesoru, tak na grafické kartě. Nejdříve je potřeba tento problém vyřešit na hlavním procesoru, protože ten s pozicemi pracuje a ukládá je. Poté je potřeba data přesunout do grafické karty a zajistit, aby nám grafická karta scénu vykreslila s dostatečnou přesnosti.

4.18.1 Floating Camera Origin

Pohybující se střed scény, anglicky nazývané jako Floating Camera Origin nebo Floating Origin je postup, při kterém se na grafickou kartu posílají upravené souřadnice. V těchto upravených souřadnicích je pozice kamery středem souřadnicového systému a proto mají souřadnice objektů nejblíže kamery největší přesnost. Aby tento algoritmus byl užitečný, musí hlavní procesor pracovat se souřadnicemi v přesnějším formátu, než jaký posíláme na grafickou kartu. Například hlavní procesor si ukládá pozice objektu ve formátu double a poté je upraví a pošle do grafické karty ve formátu float.

4.18.2 Zvetšení přesnosti pozice objektů

Dále potřebujeme zvětšit přesnost formátu, ve kterém ukládáme pozice objektů na hlavním procesoru. První volbou je použít desetinné číslo ve formátu double místo float. Ovšem i double má podobné limity jako float, tyto limity jsou pouze posunuty. Ideální by bylo použít formát, jehož přesnost je nezávislá na pozici. Takovým formátem může být například číslo s fixní desetinnou čárkou (anglicky Fixed Precision). Bohužel i číslo s fixní desetinou čárkou má své nevýhody, například nedokáže reprezentovat velmi malá čísla. Naším řešením je tedy následující datová struktura, která kombinuje výhody čísla double i čísla s fixní desetinou čárkou.

```
struct WorldPosition {

  double inSectorX, inSectorY, inSectorZ;
  long sectorX, sectorY, sectorZ;

  const double sectorSideLength = 100;

  WorldPosition Subtract(WorldPosition& other) {
    WorldPosition result;
```

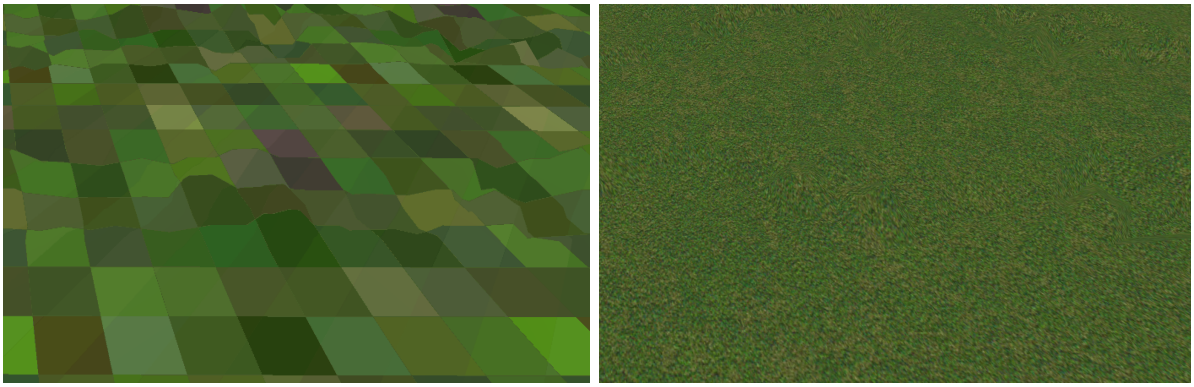
```
  result.inSectorX = this->inSectorX - other.inSectorX;
  result.inSectorY = this->inSectorY - other.inSectorY;
  result.inSectorZ = this->inSectorZ - other.inSectorZ;

  result.sectorX = this->sectorX - other.sectorX;
  result.sectorY = this->sectorY - other.sectorY;
  result.sectorZ = this->sectorZ - other.sectorZ;

  result.MoveSectorIfNeeded();
  return result;
}

WorldPosition Add(WorldPosition& other) {
  WorldPosition result;

  result.inSectorX = this->inSectorX + other.inSectorX;
  result.inSectorY = this->inSectorY + other.inSectorY;
  result.inSectorZ = this->inSectorZ + other.inSectorZ;

  result.sectorX = this->sectorX + other.sectorX;
  result.sectorY = this->sectorY + other.sectorY;
  result.sectorZ = this->sectorZ + other.sectorZ;

  result.MoveSectorIfNeeded();
  return result;
}

void MoveSectorIfNeeded() {
  long s = Math::Floor(inSectorX / sectorSideLength);
  inSectorX -= sectorSideLength * s;
  sectorX += s;

  s = Math::Floor(inSectorY / sectorSideLength);
  inSectorY -= sectorSideLength * s;
  sectorY += s;

  s = Math::Floor(inSectorZ / sectorSideLength);
  inSectorZ -= sectorSideLength * s;
  sectorZ += s;
```

```
  }
}
```

Výpis 21: Kód pro strukturu pro pozice objektů.

4.18.3 Testování a výsledky

Aplikace byla tvořena tak, aby dokázala zobrazit planetu o poloměru Země (6,371 km). S pomocí Floating Camera Origin a naší struktury pro pozice objektu je tento cíl dosažitelný. Při takto velkém poloměru planety však vyvstávají další problémy. Například triplanární mapování nebo teselační shader potřebují pracovat s pozicemi, jejichž střed souřadnicového systému je ve středu planety. V těchto případech vznikají u výpočtu s takto velkým čísly artefakty.



Obrázek 27: Ukázka vlivu poloměru planety na triplanární mapování: vlevo planeta o poloměru Země, vpravo planeta o desetině poloměru Země.

4.19 Pohyb kamery

Demonstrační aplikace obsahuje dva režimy pohybu kamery, a to: volný pohyb a pohyb po povrchu planety. Oba tyto režimy pohybu využívají jednoduché kolize s planetou.

4.19.1 Kolize s planetou

V naší ukázkové aplikaci nepoužíváme žádný fyzikální engine, jeden z důvodu je, že fyzikální enginy obvykle pracují se standartním 32 nebo 64 bitovým číslem s plovoucí desetinnou čárkou, ovšem my používáme pro pozice naši vlastní strukturu. Kolize s planetou tedy funguje na principu vrhání paprsku na vygenerované trojúhelníkové sítě, jež jsme stáhli z grafické karty. Nejdřív se pomocí vrhání paprsku směrem od kamery do středu planety na rozsahy segmentů rekurzivně zjistí, nad jakým segmentem se nacházíme. Poté se paprsek vrhne na trojúhelníkovou síť segmentu, čímž se zjistí kýžená vzdálenost od středu planety v daném směru. Všechny informace se poté převedou do zeměpisných souřadnic a porovná se jejich vzdálenost od středu planety.

Pokud je kamera pod povrchem planety, upraví se její pozice tak, aby se kamera nacházela na povrchu planety.

```
class Planet {
    float GetSurfaceHeight(vec3 position) {
        Ray rayFromPlanet = Ray(vec3(0,0,0), position);
        foreach(Segment* rootSegment in this->rootSegments)
            if(rayFromPlanet.CastRay(rootSegment->range).DidHit())
                return rootSegment.GetSurfaceHeight(position);
    }
}
class Segment {
    float GetSurfaceHeight(vec3 position) {
        position = this->TransformToSegmentLocalPosition(position);
        Ray ray = Ray(this->range.Center().Normalized(), position);
        foreach(Triangle* triangle in this->GetMeshTriangles()) {
            RayHitInfo hit = ray.CastRay(triangle);
            if (hit.DidHit())
                return ray.GetPoint(hit.HitDistance()).Length();
        }
    }
}
```

Výpis 22: Zjednodušený kód pro výpočet vzdáleností od středu planety pro libovolný bod.

4.19.2 Volný pohyb

Volný pohyb umožňuje 6 stupňů volnosti pohybu, což znamená rotaci a pohyb ve všech třech osách. Myší lze rotovat kolem dvou os, klávesami pak kolem osy třetí, a nakonec posun ve všech třech osách pomocí kláves. Rychlost pohybu je dána vzdáleností od planety a také je modifikovatelná kolečkem na myši.

```
yawDelta = mouseDelta.x;
pitchDelta = mouseDelta.y;
if (Input::GetKey(Key::Q)) rollDelta -= 1;
if (Input::GetKey(Key::E)) rollDelta += 1;

if (Input::GetKey(Key::W)) movementDirection += Constants::vec3Forward;
if (Input::GetKey(Key::S)) movementDirection -= Constants::vec3Forward;
if (Input::GetKey(Key::D)) movementDirection += Constants::vec3Right;
```

```
if (Input::GetKey(Key::A)) movementDirection -= Constants::vec3Right;
if (Input::GetKey(Key::Space)) movementDirection += Constants::vec3Up;
if (Input::GetKey(Key::ControlLeft)) movementDirection -= Constants::vec3Up;

this->cameraRotation = this->cameraRotation * (
  Math::QuatFromAxisAngle(vec3.UnitX, pitchDelta) *
  Math::QuatFromAxisAngle(vec3.UnitY, yawDelta) *
  Math::QuatFromAxisAngle(vec3.UnitZ, rollDelta);
);

this->cameraPosition +=
    movementDirection.RotateBy(this->cameraRotation) *
    timeDelta * distanceFromClosestPlanet;
```

Výpis 23: Zjednodušený kód pro výpočet rotace a posunu pro volný pohyb kamery.

4.19.3 Pohyb po povrchu planety

Druhou možností pohybu kamery je pohyb po povrchu planety. V tomto režimu je simulována gravitace, výška kamera je tedy v každém snímku upravena tak, aby byla 2 metry nad povrchem planety. Dále požadujeme, aby rotace kamery v tomto režimu byla závislá na planetě, tedy: pokud se kamera dívá na horizont planety, může se kamera bez další manuální změny rotace pohybovat stále kupředu a její rotace bude pořád směrem k horizontu. Tohoto efektu docílíme tak, že rotaci kamery v tomto režimu složíme ze dvou normalizovaných směrových vektorů. Těmito vektory jsou: vektor ze středu planety směrem ke kameře a dopředný směrový vektor. Klíčové je tento dopředný směrový vektor v každém snímku upravit podle změn směrového vektoru ze středu planety ke kameře.

```
vec3 up = planet->Center.Towards(this->position).Normalized();
vec3 forward = this->lastVectorForward;

float upDeltaAngle = up.Angle(lastVectorUp);
vec3 upDeltaRot = Math::QuatFromAxisAngle(up.Cross(lastVectorUp), upDeltaAngle)
    .Inverted();
forward = forward.RotateBy(upDeltaRot);

vec3 left = up.Cross(forward);
quat rotationDelta =
    Math::QuatFromAxisAngle(up, -yawDelta) *
    Math::QuatFromAxisAngle(left, pitchDelta);
```

```
forward = forward.RotateBy(rotationDelta).Normalized();

// clamp up/down rotation
float minUp = Math::ToRadians(10);
float maxDown = Math::ToRadians(170);
float angle = forward.Angle(up);
if (angle < minUp) forward = up.RotateBy(Math::QuatFromAxisAngle(left, minUp));
else if (angle > maxDown) forward = up.RotateBy(Math::QuatFromAxisAngle(left,
    maxDown));

this->rotation = Math::QuatLookRotation(forward, up);

this->lastVectorUp = up;
this->lastVectorForward = forward;
```

Výpis 24: Kód pro výpočet rotace pro pohyb kamery po povrchu planety.

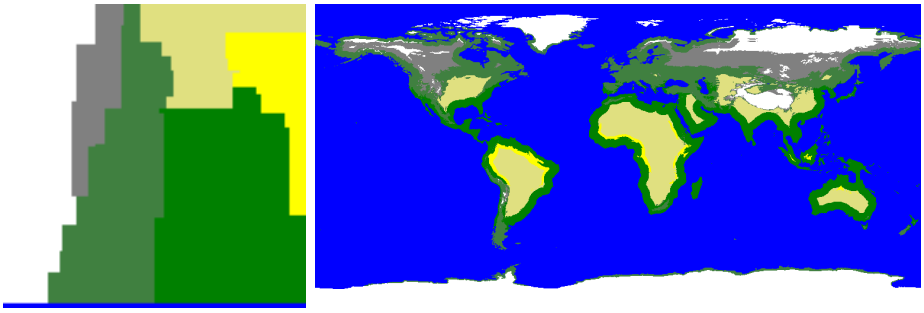
4.20 Generování a vykreslování biomů

Pro potřeby generování textur definující biomy na planetě vzhledem k zeměpisné šířce a výšce, byla vytvořena jednoúčelová externí konzolová aplikace. Vstupem do této konzolové aplikace je výšková textura a kontrolní textura biomů. Zpočátku byla výstupem této aplikace textura, která v barvě kóduje index biomu na daném místě planety. Tento přístup mapování biomů na planetu není ideální, neboť z této textury musíme číst bez interpolace a mip mapování. Takovou texturu nelze interpolovat, neboť interpolací by vznikly barvy, které nelze namapovat na naše indexy biomů. Pokud tuto texturu použijeme, budou kvůli nepřítomnosti interpolace viditelné nepěkné binární přechody biomů. Proto byla tato jednoúčelová externí konzolová aplikace modifikována tak, aby výstupem byl libovolný počet splat map, jež v každém kanálu definují intenzitu biomu na daném místě planety. Tyto splat mapy již lze téměř bez problémů interpolovat. UV souřadnice kontrolní textury biomů jsou dvě čísla definující vlhkost a teplotu. Teplota je vypočítána pomocí nadmořské výšky a vzdálenosti od pólů, a to tím způsobem, že nižší teplota je blíže pólům a ve větší nadmořské výšce. Vlhkost je vypočítána pomocí vzdálenosti od moře, čím blíže k moři, tím větší vlhkost.

```
float GetTemperature(GeographicCoords g) {
    float altFromBottomOfSea = GetProceduralHeight01(g);
    float altFromSeaLevel = Math::Clamp01(altFromBottomOfSea - this->planet->
        seaLevel);
    float distanceFromPoles = 1 - Math::Abs(g.Latitude01() - 0.5) * 2;
    float temperature = (1 - altFromSeaLevel) * distanceFromPoles;
```

```
    return temperature;
}

float GetHumidity(GeographicCoords g) {
    return 1 - GetDistanceToWater(g);
}

float GetDistanceToWater(GeographicCoords g) {
    float seaLevel = this->planet->seaLevel;
    if(this->planet->GetProceduralHeight01(g) < seaLevel)
        return 0;
    float maxDistanceToWater = 0.05;
    float distanceToWater = 0;
    int splits = 3;
    while(distanceToWater < maxDistanceToWater) {
        float angleDelta = 2 * Constants::PI / splits;
        for(float angle = 0; angle < 2 * Constants::PI; angle += angleDelta)
        {
            GeographicCoords test = g;
            test.longitude01 += Math::Cos(angle) * distanceToWater;
            test.latitude01 += Math::Sin(angle) * distanceToWater;
            if(this->planet->GetProceduralHeight01(test) < seaLevel)
                return distanceToWater / maxDistanceToWater;
        }
        distanceToWater += 0.001;
        splits += 1;
    }
    return 1;
}
```

Výpis 25: Kód pro výpočet vlhkosti a teploty.



Obrázek 28: Použitá kontrolní textura biomů založena na Whittakerově diagramu a výsledné vygenerováné biomy.

4.20.1 Testování a výsledky

V naší demonstrační aplikaci byla výška povrchu planety od jejího středu definována výškovou mapou a šumovými funkcemi. Biomy jsme v jednoúčelové externí konzolové aplikaci generovali pouze za pomoci výškové mapy. Výsledkem bylo, že v demonstrační aplikaci na některých místech byly nepřesné biomy, neboť šumové funkce na některých místech přesunuly povrch nad moře, ovšem biom zůstal nezměněn. Generování biomů tedy bylo přesunuto z jednoúčelové externí konzolová aplikace do compute shaderu v demonstrační aplikaci, biomy se pak vypočítají z opravdových finálních výšek povrchu. Data biomů byla přesunuta ze splat map do trojúhelníkových síti segmentů. Každý vrchol trojúhelníkové sítě segmentu má ve výsledku definován i svůj biom.

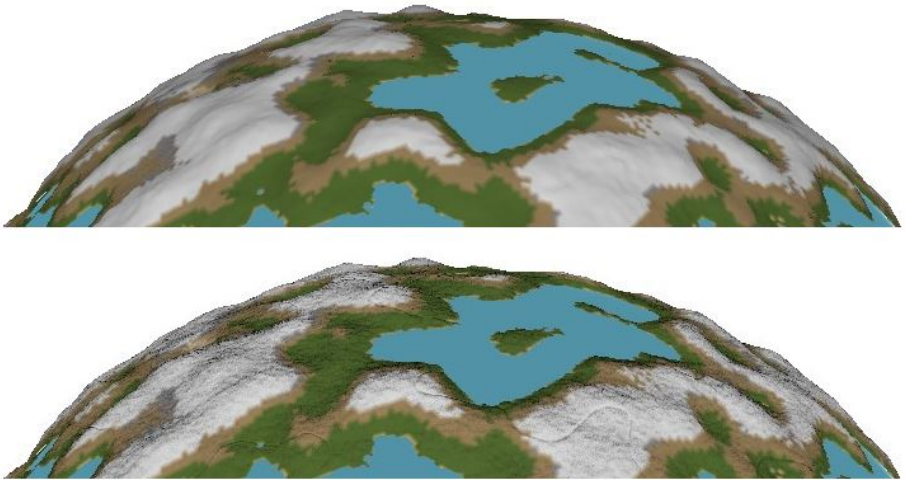
4.21 Generování normálových map segmentů

Pokud jsou normály segmentů určeny pouze pomocí jejich vrcholů, nevypadá povrch planety dostatečně detailně. Detail planety lze zlepšit použitím více vrcholů pro segmenty nebo zmenšením váhy, která je potřeba, aby se segment rozdělil na potomky. Obě předcházející řešení nám ovšem můžou negativně ovlivnit počet vykreslení snímku za sekundu, neboť přináší více trojúhelníků pro vykreslení. Nejideálnějším řešením je jednou vypočítat detailní normálovou mapu segmentu. Tento výpočet normálové mapy se provede v OpenGL compute shaderu. Normálu na jekémkoliv místě planety lze odvodit z výšek planety ve směrech tangenty a bitangenty. Ukázalo se, že generování této normálové mapy lze urychlit, pokud nastavíme Compute Shader tak, aby využíval větší work group [33].

```
dvec3 GetProceduralAndBaseHeightMapNormal(vec3 position, vec3 tangent, vec3
    bitangent, float eps) {
    dvec3 result;
    double z = GetProceduralHeight(position);
    result.x = GetProceduralHeight(position - tangent * eps) - z;
    result.y = GetProceduralHeight(position - bitangent * eps) - z;
    result.z = eps;
```

```
    return normalize(result);
}
```

Výpis 26: Kód pro výpočet normály na libovolném místě planety.



Obrázek 29: Ukázka s vypnutými a zapnutými vygenerovanými normál mapami segmentů (horní obrázek vypnuto, dolní obrázek zapnuto).

5 Závěr

V rámci této práce byly prakticky popsány některé techniky pro vykreslení, generování a optimalizaci planet. Tyto techniky byly zaměřeny zejména na možnost přiblížení planet až na vzdálenost člověka na povrchu.

V demonstrační aplikaci byl naimplementován a použit vlastní vykreslovací (herní) engine. Vizualizace demonstrační aplikace je v reálném čase a proto bylo potřeba se zaměřit také na optimalizační techniky. Optimalizace zahrnovaly především Frustrum Culling, softwarový z bufer a využití více vláken pro připravení dat na vykreslení v následujícím snímku.

Pro vykreslování planet byla do demonstrační aplikace prakticky implementována většina teoreticky popsaných algoritmů. Dále jsme v demonstrační aplikaci využili řadu dalších urychlení a zlepšení, jež nám zajistily lepší vizuální výsledek a rychlejší generování planety. Pro vykreslení planety byl implementován a otestován především algoritmus Chunked LOD [18] a výpočet šumových funkcí na grafické kartě. Naším rozšířením je například generování biomů nebo generování normálových map segmentů. Obě tyto rozšíření dopomohly k lepšímu vizuálnímu výsledku demonstrační aplikace.

Uživatel si může vyzkoušet vliv některých optimalizací na výsledný výkon demonstrační aplikace, taktéž si může měnit kontrolní texturu biomů či velikosti planet.

Z výsledku testování je znatelné, že přesunem paralelizovatelných operací na grafickou kartu, či chytrým připravením dat v samostatném vlákně, zatímco hlavní vlákno právě vykresluje, lze dosáhnout většího počtu překreslení snímků za sekundu a zkrácení doby, po kterou trvá povrch planety vygenerovat.

Při tvorbě této práce se objevila spousta dalších možností pro vylepšení vizuálního výsledku a výkonu aplikace. Vždy se lze zaměřit na lepší vzhled, modely a textury biomů. Dalším zajímavým rozšířením může být například vyřešení simulace fyziky a gravitace ve virtuálním světe, v němž přesnost standartního desetinného čísla float nestačí. Dále by bylo zajímavé simulovat realistický pohyb planet v celé sluneční soustavě.

Literatura

[1] SHAKER, Noor. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* TOGELIUS, Julian; NELSON, Mark J. Springer. 2016. ISBN 978-3-319-42714-0.

[2] TOGELIUS, J. KASTBJERG, E., SCHEDL, D., YANNAKAKIS, G.N. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd Workshop on Procedural Content Generation in Games* IT University of Copenhagen, Copenhagen, Denmark. NY, USA. ACM New York. 2011. ISBN 978-1-4503-0872-4.

[3] PLANETSIDE, Software. *Terragen* [online]. 2017 [cit. 2017-4-28] . Dostupné z: `http://planetside.co.uk`

[4] E-ON, Software. *E-on Vue* [online]. 2017 [cit. 2017-4-28] . Dostupné z: `http://www.e-onsoftware.com`

[5] WORLD MACHINE SOFTWARE, LLC. *WorldMachine* [online]. 2014 [cit. 2017-4-28] . Dostupné z: `http://www.world-machine.com`

[6] RAK Jakub. *Procedurální generování terénu.* Ostrava, 2014. Bakalářská práce na Fakultě elektrotechniky a informatiky Technické univerzity Ostrava na katedře Informatiky a výpočetní techniky. Vedoucí bakalářské práce Ing. Martin Němec Ph.D.

[7] MICHAEL C. Toy, KENNETH C. R. C. Arnold *Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California: A Guide to the Dungeons of Doom* [online]. 1980. [cit.2017-01-15]. Dostupné z: `https://docs.freebsd.org/44doc/usd/30.rogue/paper.pdf`

[8] TOM HATFIELD. *Jan 29, 2013* [online]. Rise Of The Roguelikes: A Genre Evolves [cit. 2017-4-28] . Dostupné z: `http://pc.gamespy.com/pc/ftl-faster-than-light/1227287p1.html`

[9] POHLMANN, Kenneth C. *The Compact Disc Handbook.* Middleton, Wisconsin: A-R Editions. 1992. 2nd edition. ISBN 0-89579-300-8.

[10] IDV, Inc. *SpeedTrree* [online]. 2017 [cit. 2017-4-28] . Dostupné z: `http://www.speedtree.com/`

[11] GAMES BY FRONTIER. *Elite* [online]. 2010 [cit. 2017-4-28] . Dostupné z: `http://frontier.co.uk/games/elite`

[12] BETHESDA, Softworks LLC. *The Elder Scrolls Official Site | The Elder Scrolls: Arena* [online]. c2017. [cit.2017-03-20]. Dostupné z: `https://elderscrolls.bethesda.net/arena/`

[13] TASAJÄRVI, Lassi. *Demo Scene: the Art of Real-Time. Evenlake Studios.* 2004. ISBN 952-91-7022-X.

[14] .THEPRODUKKT. *Free .kkrieger Download* [online]. 2006. [cit.2017-04-10]. Dostupné z: `http://www.acid-play.com/download/kkrieger`

[15] QUIXEL. *Quixel Suite Manual* [online]. © 2016, [cit.2017-01-17]. Dostupné z: `http://quixel.se/usermanual/quixelsuite/doku.php`

[16] ALLEGORITHMIC. *Documentation for the Substance products* [online]. © 2017, [cit.2017-04-17]. Dostupné z: `https://docs.allegorithmic.com/documentation/`

[17] HEBÁK, Petr a Jana KAHOUNOVÁ. *Počet pravděpodobnosti v příkladech.* 7., nezměn. vyd. Praha: Informatorium, 2014. ISBN 978-80-7333-109-2.

[18] THOMAS LAURITSEN & STEEN LUND NIELSEN. *Rendering Very Large Very Detailed Terrains* [online]. 2005 [cit. 2017-4-28] . Dostupné z: `http://www.terrain.dk/terrain.pdf`

[19] HENRY VUONTISJÄRVI, Tietotekniikan koulutusohjelma Oulun seudun ammattikorkeakoulu. *Jaro 2014* [online]. PROCEDURAL PLANET GENERATION IN GAME DEVELOPMENT [cit. 2017-4-28] . Dostupné z: `https://www.theseus.fi/bitstream/handle/10024/82203/opinnaytetyo.pdf`

[20] LENGYEL, Eric. *The Transvoxel^{TM} Algorithm* [online]. 2010 [cit. 2017-4-28] . Dostupné z: `http://www.terathon.com/voxels`

[21] WHITTAKER, R. H. *Communities and Ecosystems.* Edition 2, illustrated. 1975. ISBN 10: 0024273902. ISBN 13: 9780024273901.

[22] Blanco-Canqui, Humberto; Rattan, Lal (2008). "Soil and water conservation". Principles of soil conservation and management. Dordrecht: Springer. pp. 1–20. ISBN 9781402087097.

[23] JACOB OLSEN, University of Southern Denmark, Department of Mathematics And Computer Science. *Realtime Procedural Terrain Generation* [online]. 2004 [cit. 2017-4-28] . Dostupné z: `http://web.mit.edu/cesium/Public/terrain.pdf`

[24] STAVA O., BENES B., BRISBIN M. AND KRIVANEK J., Czech Technical University in Prague. *2008* [online]. Interactive Terrain Modeling Using Hydraulic Erosion [cit. 2017-4-28] . Dostupné z: `http://cgg.mff.cuni.cz/~jaroslav/papers/2008-sca-erosim/2008-sca-erosiom-fin.pdf`

[25] PAUL KNIGHT, Diplograph. [online]. Hardware Linear Feedback Shift Registers in the NES [cit. 2017-4-28] . Dostupné z: `https://diplograph.net/posts/hardware_linear_feedback_shift_registers_in_the_nes`

[26] Game Development Essentials: Game QA & Testing, By Luis Levy, Jeannie Novak, ISBN-13: 978-1-4354-3947-4

[27] NVIDIA. *GPU gems* [online]. 2004 [cit. 2017-4-28] . Dostupné z: `http://http.developer.nvidia.com/GPUGems/gpugems_part01.html`

[28] NVIDIA. *GPU gems 2* [online]. 2005 [cit. 2017-4-28] . Dostupné z: `http://http.developer.nvidia.com/GPUGems2/gpugems2_part01.html`

[29] BITTNER, J., M. WIMMER, H. PIRINGER, AND W. PURGATHOFER., Computer Graphics Forum (Proceedings of Eurographics). *2004* [online]. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful [cit. 2017-4-28] .

[30] DANIEL COLLIN (DICE), GDC. *2013* [online]. Culling the Battlefield [cit. 2017-4-28] . Dostupné z: `http://www.frostbite.com/wp-content/uploads/2013/05/CullingTheBattlefield.pdf`

[31] TOMAS, Akenine-Möller, *Real-Time Rendering*, ERIC Haines, NATY Hoffman, Third Edition, ISBN-13: 978-1-4987-8563-1

[32] UNITY, Technologies. *Unity Game Engine* [online]. 2014 [cit. 2017-4-28] . Dostupné z: `https://unity3d.com`

[33] OpenGL Software Development Kit. *OpenGL - The Industry Standard for High Performance Graphics [online]*. Beaverton, USA: Khronos Group, © 1997-2015 [cit. 2017-04-17]. Dostupné z: `https://www.opengl.org/sdk/`

[34] 3D Engine Design for Virtual Globes, Patrick Cozzi, Kevin Ring. ISBN: 978-1-5688-711-8

Seznam příloh

A Výsledné obrázky z demonstrační aplikace

B Elektronická verze práce — Příloha na DVD
- \Documents\Thesis.pdf

C Spustitelná demonstrační aplikace — Příloha na DVD
- \Application\Release\bin\MyGame.exe

D Zdrojové kódy demonstrační Aplikace — Příloha na DVD
- \Application\Source

A Výsledné obrázky z demonstrační aplikace